



# UNITED STATES PATENT AND TRADEMARK OFFICE

UNITED STATES DEPARTMENT OF COMMERCE  
United States Patent and Trademark Office  
Address: COMMISSIONER FOR PATENTS  
P.O. Box 1450  
Alexandria, Virginia 22313-1450  
www.uspto.gov

APPLICATION NO.	FILING DATE	FIRST NAMED INVENTOR	ATTORNEY DOCKET NO.	CONFIRMATION NO.
10/757,227	01/14/2004	Jimmie Earl DeWitt JR.	AUS92003055US1	3250

35525 7590 02/28/2006

IBM CORP (YA)  
C/O YEE & ASSOCIATES PC  
P.O. BOX 802333  
DALLAS, TX 75380



EXAMINER

SAVLA, ARPAN P

ART UNIT PAPER NUMBER

2185

DATE MAILED: 02/28/2006

Please find below and/or attached an Office communication concerning this application or proceeding.

BEST AVAILABLE COPY

<b>Office Action Summary</b>	<b>Application No.</b>	<b>Applicant(s)</b>	
	10/757,227	DEWITT ET AL.	
	<b>Examiner</b>	<b>Art Unit</b>	
	Arpan P. Savla	2185	

-- The MAILING DATE of this communication appears on the cover sheet with the correspondence address --

#### Period for Reply

A SHORTENED STATUTORY PERIOD FOR REPLY IS SET TO EXPIRE 3 MONTH(S) OR THIRTY (30) DAYS, WHICHEVER IS LONGER, FROM THE MAILING DATE OF THIS COMMUNICATION.

- Extensions of time may be available under the provisions of 37 CFR 1.136(a). In no event, however, may a reply be timely filed after SIX (6) MONTHS from the mailing date of this communication.
- If NO period for reply is specified above, the maximum statutory period will apply and will expire SIX (6) MONTHS from the mailing date of this communication.
- Failure to reply within the set or extended period for reply will, by statute, cause the application to become ABANDONED (35 U.S.C. § 133). Any reply received by the Office later than three months after the mailing date of this communication, even if timely filed, may reduce any earned patent term adjustment. See 37 CFR 1.704(b).

#### Status

- 1) ☒ Responsive to communication(s) filed on 14 January 2004.
- 2a) ☐ This action is FINAL. 2b) ☒ This action is non-final.
- 3) ☐ Since this application is in condition for allowance except for formal matters, prosecution as to the merits is closed in accordance with the practice under *Ex parte Quayle*, 1935 C.D. 11, 453 O.G. 213.

#### Disposition of Claims

- 4) ☒ Claim(s) 1-21 is/are pending in the application.
- 4a) Of the above claim(s) \_\_\_\_\_ is/are withdrawn from consideration.
- 5) ☐ Claim(s) \_\_\_\_\_ is/are allowed.
- 6) ☒ Claim(s) 1-21 is/are rejected.
- 7) ☐ Claim(s) \_\_\_\_\_ is/are objected to.
- 8) ☐ Claim(s) \_\_\_\_\_ are subject to restriction and/or election requirement.

#### Application Papers

- 9) ☒ The specification is objected to by the Examiner.
- 10) ☒ The drawing(s) filed on 14 January 2004 is/are: a) ☒ accepted or b) ☐ objected to by the Examiner.  
Applicant may not request that any objection to the drawing(s) be held in abeyance. See 37 CFR 1.85(a).  
Replacement drawing sheet(s) including the correction is required if the drawing(s) is objected to. See 37 CFR 1.121(d).
- 11) ☐ The oath or declaration is objected to by the Examiner. Note the attached Office Action or form PTO-152.

#### Priority under 35 U.S.C. § 119

- 12) ☐ Acknowledgment is made of a claim for foreign priority under 35 U.S.C. § 119(a)-(d) or (f).
- a) ☐ All b) ☐ Some \* c) ☐ None of:
- ☐ Certified copies of the priority documents have been received.
  - ☐ Certified copies of the priority documents have been received in Application No. \_\_\_\_\_.
  - ☐ Copies of the certified copies of the priority documents have been received in this National Stage application from the International Bureau (PCT Rule 17.2(a)).

\* See the attached detailed Office action for a list of the certified copies not received.

#### Attachment(s)

- |  |   |
|--|---|
| 1) <input checked="" type="checkbox"/> Notice of References Cited (PTO-892)  | 4) <input type="checkbox"/> Interview Summary (PTO-413)<br>Paper No(s)/Mail Date. _____ |
| 2) <input type="checkbox"/> Notice of Draftsperson's Patent Drawing Review (PTO-948)   | 5) <input type="checkbox"/> Notice of Informal Patent Application (PTO-152)             |
| 3) <input checked="" type="checkbox"/> Information Disclosure Statement(s) (PTO-1449 or PTO/SB/08)<br>Paper No(s)/Mail Date <u>1/14/04, 7/1/05</u> | 6) <input type="checkbox"/> Other: _____  |

Art Unit: 2185

### **DETAILED ACTION**

The instant application having Application No. 10/757,227 has a total of 21 claims pending in the application, there are 3 independent claims and 18 dependent claims, all of which are ready for examination by Examiner.

### **INFORMATION CONCERNING OATH/DECLARATION**

#### **Oath/Declaration**

1. Applicant's oath/declaration has been reviewed by Examiner and is found to conform to the requirements prescribed in 37 CFR 1.63.

### **INFORMATION CONCERNING DRAWINGS**

#### **Drawings**

2. Applicant's drawings submitted January 14, 2004 are acceptable for examination purposes.

### **ACKNOWLEDGMENT OF REFERENCES CITED BY APPLICANT**

#### **Information Disclosure Statement**

3. As required by MPEP § 609(c), Applicant's submission of both Information Disclosure Statements dated January 14, 2004 and July 1, 2005 are acknowledged by Examiner and cited references have been considered in the examination of the claims now pending. As required by MPEP § 609 c(2), a copy of the PTOL-1449 initialed and dated by Examiner is attached to the instant office action.

Art Unit: 2185

## **OBJECTIONS**

### **Specification**

4. The disclosure is objected to because of the following informalities:
5. The lengthy specification has not been checked to the extent necessary to determine the presence of all possible minor errors. Applicant's cooperation is requested in correcting any errors of which applicant may become aware in the specification.
6. In the section entitled "Cross Reference to Related Applications" Applicant must properly identify all co-pending applications with their corresponding application numbers (i.e. serial numbers).

Appropriate correction is required.

## **REJECTIONS NOT BASED ON PRIOR ART**

### **Claim Rejections - 35 USC § 101**

7. 35 U.S.C. 101 reads as follows:

Whoever invents or discovers any new and useful process, machine, manufacture, or composition of matter, or any new and useful improvement thereof, may obtain a patent therefor, subject to the conditions and requirements of this title.

8. **Claims 11-20 are rejected under 35 U.S.C. 101 because the claimed invention is directed to non-statutory subject matter.** Claims 11-20 are not limited to tangible embodiments. In view of Applicant's disclosure, pg. 125, line 23 – pg. 126, line 11, the computer readable medium is not limited to tangible embodiments, instead being define as including both tangible embodiments (e.g. recordable-type media, such as a floppy disk, hard disk drive, a RAM, CD-ROMS, and DVD-ROMS) and intangible

Art Unit: 2185

embodiments (e.g. transmission forms, such as radio frequency and light wave transmissions). As such, claims 11-20 are not limited to statutory subject matter and are therefore non-statutory.

**Claim Rejections - 35 USC § 112**

9. The following is a quotation of the second paragraph of 35 U.S.C. 112:

The specification shall conclude with one or more claims particularly pointing out and distinctly claiming the subject matter which the applicant regards as his invention.

10. **Claims 1-21 are rejected under 35 U.S.C. 112, second paragraph, as being indefinite for failing to particularly point out and distinctly claim the subject matter which applicant regards as the invention.**

11. **As per claims 1, 3, 11, 13, and 21**, the claims recite the limitation "the contents of the cache line" in lines 11, 2, 13, 2, and 11 respectively. There is insufficient antecedent basis for this limitation in the claims. Applicant may consider amending the claims to read "a contents of the cache line."

12. **Claims 3 and 13 are rejected under 35 U.S.C. 112, second paragraph, as being indefinite for failing to particularly point out and distinctly claim the subject matter which applicant regards as the invention.**

13. **As per claims 3 and 13**, the claims recites the limitation "the code of the computer program" in lines 6 and 7 respectively. There is insufficient antecedent basis for this limitation in the claims. Applicant may consider amending the claims to read "code of the computer program."

Art Unit: 2185

14. **Claims 5-10 and 15-20** are rejected under 35 U.S.C. 112, second paragraph, as being indefinite for failing to particularly point out and distinctly claim the subject matter which applicant regards as the invention.

15. **As per claims 5, 6, 10, 15, 16, and 20**, the claims recite the limitation "the reload operation" in lines 10, 2, 3, 10, 2, and 4 respectively. There is insufficient antecedent basis for this limitation in the claims. Applicant may consider amending the claims to read "the reload."

16. **Claims 6-9 and 16-19** are rejected under 35 U.S.C. 112, second paragraph, as being indefinite for failing to particularly point out and distinctly claim the subject matter which applicant regards as the invention.

17. **As per claims 6, 7, 16, and 17**, it is unclear whether the "value of the first processor flag bit" recited in lines 10, 7, 11, and 9 respectively is the same as the "first value of the first processor flag bit" recited in lines 4, 2, 4, and 3 respectively or a completely different "value of the first processor flag bit." Applicant may consider amending "first value of the first processor flag bit" to read "value of the first processor flag bit" instead.

### **Conclusion**

### **STATUS OF CLAIMS IN THE APPLICATION**

The following is a summary of the treatment and status of all claims in the application as recommended by MPEP 707.70(i):

**Allowable Subject Matter**

18. **Claims 1-10 and 21** would be allowable if rewritten or amended to overcome the rejection(s) under 35 U.S.C. 112, 2nd paragraph, set forth in this Office action.

19. **Claims 11-20** would be allowable if rewritten or amended to overcome the rejection(s) under 35 U.S.C. 101 set forth in this Office action.

20. The primary reasons for allowance of **claims 1-21** in the instant application is the combination with the inclusion in these claims that **"identifying false cache line sharing during execution of a computer program** in a multiprocessor data processing system, comprising: **determining whether the cache line is being falsely shared between processors based on values of the processor flag bits."** The prior art of record neither anticipates nor renders obvious the above recited combination.

21. As allowable subject matter has been indicated, applicant's response must either comply with all formal requirements or specifically traverse each requirement not complied with. See 37 C.F.R. § 1.111(b) and § 707.07(a) of the MPEP.

**RELEVANT ART CITED BY THE EXAMINER**

The following prior art made of record and not relied upon is cited to establish the level of skill in Applicant's art and those arts considered reasonably pertinent to Applicant's disclosure. See MPEP 707.05(e).

22. U.S. Patent 5,710,881 discloses a data merging method and apparatus for shared memory multiprocessing computer systems.

Art Unit: 2185

23. U.S. Patent 5,822,763 discloses cache coherence protocol for reducing the effects of false sharing in non-bus-based shared-memory multiprocessors.
24. U.S. Patent 5,928,334 discloses a method for detecting synchronization violations in a multiprocessor computer system.
25. U.S. Patent 6,094,709 discloses a method of reducing false sharing in a shared memory system by enabling two caches to modify the same line at the same time.
26. U.S. Patent 6,285,974 discloses a method for detecting architectural violations in a multiprocessor computer system.
27. U.S. Patent 6,636,950 discloses computer architecture for shared memory access.
28. U.S. Patent Application Publication 2004/0205302 discloses a method and system for postmortem identification of falsely shared objects.

#### **Non-Patent Literature**

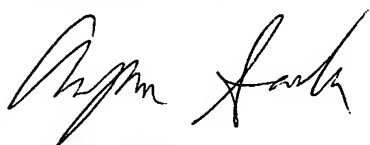
Torrellas et al., "False Sharing and Spatial Locality in Multiprocessor Caches", June 1994, IEEE Transactions on Computers, Vol. 43, No. 6, pp. 651-663.

Rothman et al., "Analysis of Shared Memory Misses and Reference Patterns", 2000, IEEE, pp. 187-198.

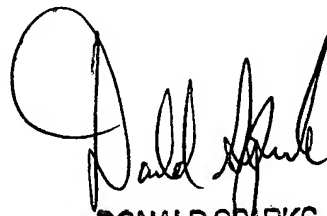
Any inquiry concerning this communication or earlier communications from the examiner should be directed to Arpan P. Savla whose telephone number is (571) 272-1077. The examiner can normally be reached on M-F 8:30-5:00.

If attempts to reach the examiner by telephone are unsuccessful, the examiner's supervisor, Donald Sparks can be reached on (571) 272-4201. The fax phone number for the organization where this application or proceeding is assigned is 571-273-8300.

Information regarding the status of an application may be obtained from the Patent Application Information Retrieval (PAIR) system. Status information for published applications may be obtained from either Private PAIR or Public PAIR. Status information for unpublished applications is available through Private PAIR only. For more information about the PAIR system, see <http://pair-direct.uspto.gov>. Should you have questions on access to the Private PAIR system, contact the Electronic Business Center (EBC) at 866-217-9197 (toll-free).



Arpan Savla  
Assistant Examiner  
Art Unit 2185  
February 14, 2006



DONALD SPARKS  
SUPERVISORY PATENT EXAMINER

<b>Form PTO-1449</b>  <b>LIST OF PRIOR ART CITED BY APPLICANT</b> <i>(Use several sheets if necessary)</i>			ATTORNEY DOCKET NO. <b>AUS920030555US1</b>		SERIAL NO. <del>Not Assigned</del> 10/757, 227	
			APPLICANT <b>DeWitt, Jr. et al.</b>			
			FILING DATE 1/14/04		GROUP ART UNIT <del>Not Assigned</del> 2085	
U.S. PATENT DOCUMENTS						
EXAMINER INITIAL	DOCUMENT NO.	PUBLICATION DATE	INVENTOR NAME	CLASS/SUBCLASS	FILING DATE	
A.S.	AA 5,103,394	Apr. 7, 1992	Blasciak	395/575	Dec. 21, 1989	
A.S.	AB 6,330,662 B1	Dec. 11, 2001	Patel et al.	712/236	Feb. 23, 1999	
A.S.	AC 6,480,938 B2	Nov. 12, 2002	Vondran, Jr.	711/125	Dec. 15, 2000	
A.S.	AD 6,430,741 B1	Aug. 6, 2002	Mattson, Jr. et al.	717/154	Feb. 26, 1999	
A.S.	AE 6,189,141 B1	Feb. 13, 2001	Benitez et al.	717/4	May 4, 1998	
A.S.	AF 5,930,508	Jul. 27, 1999	Faraboschi et al.	395/706	Jun. 9, 1997	
A.S.	AG 6,351,844 B1	Feb. 26, 2002	Bala	717/4	Nov. 5, 1998	
A.S.	AH 6,324,689 B1	Nov. 27, 2001	Lowney et al.	717/9	Sep. 30, 1998	
A.S.	AI 6,442,585 B1	Aug. 27, 2002	Dean et al.	709/108	Nov. 26, 1997	
A.S.	AJ 5,774,724	Jun. 30, 1998	Heisch	395/704	Nov. 20, 1995	
A.S.	AK 5,987,250	Nov. 16, 1999	Subrahmanyam	395/704	Aug. 21, 1997	
A.S.	AL 6,192,513 B1	Feb. 20, 2001	Subrahmanyam	717/5	Nov. 2, 1998	
A.S.	AM 5,691,920	Nov. 25, 1997	Levine et al.	364/551.01	Oct. 2, 1995	
A.S.	AN 6,223,338 B1	Apr. 24, 2001	Smolders	717/4	Sep. 30, 1998	
A.S.	AO 6,101,524	Aug. 8, 2000	Choi et al.	709/102	Oct. 23, 1997	
A.S.	AP 6,256,775 B1	Jul. 3, 2001	Flynn	717/4	Dec. 11, 1997	
A.S.	AQ 6,446,029 B1	Sep. 3, 2002	Davidson et al.	702/186	Jun. 30, 1999	
A.S.	AR 6,134,676	Oct. 17, 2000	VanHuben et al.	714/39	Apr. 30, 1998	
A.S.	AS 5,937,437	Aug. 10, 1999	Roth et al.	711/202	Oct. 28, 1996	
A.S.	AT 6,243,804 B1	Jun. 5, 2001	Cheng	712/228	Jul. 22, 1998	
A.S.	AU 4,291,371	Sep. 22, 1981	Holtey	364/200	Jan. 2, 1979	
A.S.	AV 5,938,778	Aug. 17, 1999	John, Jr. et al.	714/45	Nov. 10, 1997	
A.S.	AW 6,286,132 B1	Sep. 4, 2001	Tanaka et al.	717/4	Jan. 7, 1999	
A.S.	AX 6,206,584 B1	Mar. 27, 2001	Hastings	395/183.11	May 31, 1995	
A.S.	AY 6,374,364 B1	Apr. 16, 2002	McElroy et al.	714/10	Jan. 19, 1999	
A.S.	AZ 6,070,009	May 30, 2000	Dean et al.	395/704	Nov. 26, 1997	
A.S.	BA 5,966,537	Oct. 12, 1999	Ravichandran	395/709	May 28, 1997	
DATE CONSIDERED 2/7/06			EXAMINER			
EXAMINER: Initial if reference considered, whether or not citation is in conformance with MPEP § 609; draw line through citation if not in conformance and not considered. Include copy of this form with next communication to applicant.						

<b>Form PTO-1449</b>  <b>LIST OF PRIOR ART CITED BY APPLICANT</b> <i>(Use several sheets if necessary)</i>			ATTORNEY DOCKET NO. <b>AUS920030555US1</b>		SERIAL NO. <del>Not Assigned</del>  16/757, 227	
			APPLICANT <b>DeWitt, Jr. et al.</b>			
			FILING DATE <b>1/14/04</b>		GROUP ART UNIT <del>Not Assigned</del> <b>2185</b>	
<b>U.S. PATENT DOCUMENTS</b>						
EXAMINER INITIAL	DOCUMENT NO.	PUBLICATION DATE	INVENTOR NAME		CLASS/ SUBCLASS	FILING DATE
A.S.	BB 2002/0129309	Sep. 12, 2002	Floyd et al.		714/724	Dec. 18, 2000
A.S.	BC 2001/0032305	Oct. 18, 2001	Barry		712/34	Feb. 23, 2001
A.S.	BD 2002/0199179	Dec. 26, 2002	Lavery et al.		717/158	Jun. 21, 2001
A.S.	BE 2002/0124237	Sep. 5, 2002	Sprunt et al.		717/127	Dec. 29, 2000
A.S.	BF 2002/0147965	Oct. 10, 2002	Swaine et al.		717/124	Feb. 1, 2001
A.S.	BG 2002/0019976	Feb. 14, 2002	Patel et al.		717/137	May 25, 2001
<b>FOREIGN PATENT DOCUMENTS</b>						
EXAMINER INITIAL	DOCUMENT NO.	PUBLICATION DATE	COUNTRY		CLASS/ SUBCLASS	TRANSLATION YES NO
A.S.	BH JP2000029731	Dec. 8, 1999	Japan		G06F 9/38	X
A.S.	BI JP2000347863	Dec. 15, 2000	Japan		G06F 9/38	X
<b>OTHER PRIOR ART (including author, title, date, pertinent page, etc.)</b>						
	B.I	Kikuchi, "Parallelization Assist System", Joho Shori, Vol. 34, No. 9, Sept. 1993, pp. 1158-1169.				
A.S.	BK	Cohen et al., "Hardware-Assisted Characterization of NAS Benchmarks", Cluster Computing, Vol. 4, No. 3, July 2001, pp. 189-196.				
A.S.	BL	Talla et al., "Evaluating Signal Processing and Multimedia Applications on SIMD, VLIW and Super Scalar Architectures", International Conference on Computer Design, Austin, Sept. 17-20, 2000, pp. 163-172.				
A.S.	BM	Iwasawa et al., "Parallelization Method of Fortran DO Loops by Parallelizing Assist System", Transactions of Information Processings Society of Japan, Vol. 36, No. 8, Aug. 1995, pp. 1995-2006.				
A.S.	BN	Talla et al., "Execution Characteristics of Multimedia Applications on a Pentium II Processor", IEEE International Performance, Computing, and Communications Conference, 19 <sup>th</sup> , Phoenix, Feb. 20-22, 2000, pp. 516-524.				
A.S.	BO	IBM Research Disclosure Bulletin 444188, "Enable Debuggers as an Objective Performance Measurement Tool for Software Development Cost Reduction", April 2001, pp. 686-688.				
<b>RELATED PATENT APPLICATIONS</b>						
EXAMINER INITIAL	APPLICATION NO/ ATTY. DOCKET NO.	APPLICANT	TITLE			FILING DATE
A.S.	BP 09/435,069 AT9-99-491	Davidson et al.	Method and Apparatus for Instruction Sampling for Performance Monitoring and Debug			Nov. 4, 1999
A.S.	BQ 08/538,071 AUT919950055US1	Gover et al.	Method and System for Selecting and Distinguishing an Event Sequence using an Effective Address in a Processing System			Oct. 2, 1995
DATE CONSIDERED <b>2/7/06</b>			EXAMINER <b>Arpan Sirla</b>			
EXAMINER: Initial if reference considered, whether or not citation is in conformance with MPEP § 609; draw line through citation if not in conformance and not considered. Include copy of this form with next communication to applicant.						



U.S. DEPT. OF COMMERCE PATENT AND TRADEMARK OFFICE

Page 1 of 1

LIST OF PRIOR ART CITED BY APPLICANT (Use several sheets if necessary)	ATTORNEY DOCKET NO. AUS920030555US1	SERIAL NO. 10/757,227
	APPLICANT DeWitt, Jr. et al.	
	FILING DATE January 14, 2004	GROUP ART UNIT <del>2488</del> 2485

## RELATED PATENT APPLICATIONS

EXAMINER INITIAL		APPLICATION NO./ ATTY. DOCKET NO.	APPLICANT	TITLE	FILING DATE
A.S.	AA	10/675,777 / AUS920030477US1	DeWitt, Jr. et al.	Method and Apparatus for Counting Instruction Execution and Data Accesses	Sep. 30, 2003
A.S.	AB	10/674,604 / AUS920030478US1	DeWitt, Jr. et al.	Method and Apparatus for Selectively Counting Instructions and Data Accesses	Sep. 30, 2003
A.S.	AC	10/675,831 / AUS920030479US1	DeWitt, Jr. et al.	Method and Apparatus for Generating Interrupts Upon Execution of Marked Instructions and Upon Access to Marked Memory Locations	Sep. 30, 2003
A.S.	AD	10/675,778 / AUS920030480US1	DeWitt, Jr. et al.	Method and Apparatus for Counting Data Accesses and Instruction Executions that Exceed a Threshold	Sep. 30, 2003
A.S.	AE	10/675,776 / AUS920030481US1	DeWitt, Jr. et al.	Method and Apparatus for Counting Execution of Specific Instructions and Accesses to Specific Data Locations	Sep. 30, 2003
A.S.	AF	10/675,751 / AUS920030482US1	DeWitt, Jr. et al.	Method and Apparatus for Debug Support for Individual Instructions and Memory Locations	Sep. 30, 2003
A.S.	AG	10/675,721 / AUS920030483US1	Levine et al.	Method and Apparatus to Autonomically Select Instructions for Selective Counting	Sep. 30, 2003
A.S.	AH	10/674,642 / AUS920030484US1	Levine et al.	Method and Apparatus to Autonomically Count Instruction Execution for Applications	Sep. 30, 2003
A.S.	AI	10/674,606 / AUS920030485US1	Levine et al.	Method and Apparatus to Autonomically Take an Execution on Specified Instructions	Sep. 30, 2003
A.S.	AJ	10/675,783 / AUS920030486US1	Levine et al.	Method and Apparatus to Autonomically Profile Applications	Sep. 30, 2003
A.S.	AK	10/675,872 / AUS920030487US1	DeWitt, Jr. et al.	Method and Apparatus for Counting Instruction and Memory Location Ranges	Sep. 30, 2003
A.S.	AL	10/757,250 / AUS920030488US1	Levine et al.	Method and Apparatus for Maintaining Performance Monitoring Structures in a Page Table for use in Monitoring Performance of a Computer Program	Jan. 14, 2004
A.S.	AM	10/757,192 / AUS920030543US1	DeWitt, Jr. et al.	Method and Apparatus for Providing Pre and Post Handlers for Recording Events	Jan. 14, 2004
A.S.	AN	10/757,249 / AUS920030554US1	DeWitt, Jr. et al.	Method and Apparatus for Identifying False Cache Line Sharing	Jan. 14, 2004
A.S.	AO	10/757,197 / AUS920030556US1	DeWitt, Jr. et al.	Method and Apparatus for Optimizing Code Execution Using Annotated Trace Information having Performance Indicator and Counter Information	Jan. 14, 2004

DATE CONSIDERED 2/7/06

EXAMINER Arpan Sarla

EXAMINER: Initial if reference considered, whether or not citation is in conformance with MPEP § 609; draw line through citation if not in conformance and not considered. Include copy of this form with next communication to applicant.

<b>Notice of References Cited</b>	Application/Control No. 10/757,227		Applicant(s)/Patent Under Reexamination DEWITT ET AL.	
	Examiner Arpan P. Savla		Art Unit 2185	Page 1 of 1

**U.S. PATENT DOCUMENTS**

*		Document Number Country Code-Number-Kind Code	Date MM-YYYY	Name	Classification
*	A	US-5,710,881	01-1998	Gupta et al.	709/200
*	B	US-5,822,763	10-1998	Baylor et al.	711/141
*	C	US-5,928,334	07-1999	Mandyam et al.	709/248
*	D	US-6,094,709	07-2000	Baylor et al.	711/141
*	E	US-6,285,974	09-2001	Mandyam et al.	703/13
*	F	US-6,636,950	10-2003	Mithal et al.	711/141
*	G	US-2004/0205302	10-2004	Cantrill, Bryan	711/141
	H	US-			
	I	US-			
	J	US-			
	K	US-			
	L	US-			
	M	US-			

**FOREIGN PATENT DOCUMENTS**

*		Document Number Country Code-Number-Kind Code	Date MM-YYYY	Country	Name	Classification
	N					
	O					
	P					
	Q					
	R					
	S					
	T					

**NON-PATENT DOCUMENTS**

*		Include as applicable: Author, Title Date, Publisher, Edition or Volume, Pertinent Pages)
	U	Torrellas et al., "False Sharing and Spatial Locality in Multiprocessor Caches", June 1994, IEEE Transactions on Computers, Vol. 43, No. 6, pp. 651-663.
	V	Rothman et al., "Analysis of Shared Memory Misses and Reference Patterns", 2000, IEEE, pp. 187-198.
	W	
	X	

\*A copy of this reference is not being furnished with this Office action. (See MPEP § 707.05(a).)  
Dates in MM-YYYY format are publication dates. Classifications may be US or foreign.

# False Sharing and Spatial Locality in Multiprocessor Caches

Josep Torrellas, *Member, IEEE*, Mónica S. Lam, *Member, IEEE*, and John L. Hennessy, *Fellow, IEEE*

**Abstract**—The performance of the data cache in shared-memory multiprocessors has been shown to be different from that in uniprocessors. In particular, cache miss rates in multiprocessors do not show the sharp drop typical of uniprocessors when the size of the cache block increases. The resulting high cache miss rate is a cause of concern, since it can significantly limit the performance of multiprocessors. Some researchers have speculated that this effect is due to false sharing, the coherence transactions that result when different processors update different words of the same cache block in an interleaved fashion. While the analysis of six applications in this paper confirms that false sharing has a significant impact on the miss rate, the measurements also show that poor spatial locality among accesses to shared data has an even larger impact. To mitigate false sharing and to enhance spatial locality, we optimize the layout of shared data in cache blocks in a programmer-transparent manner. We show that this approach can reduce the number of misses on shared data by about 10% on average.

**Index Terms**—Multiprocessing, shared-memory multiprocessor, cache memory, sharing, false sharing, optimizing compiler, placement of data.

## 1. INTRODUCTION

SCALABLE machines that support a shared-memory paradigm are a promising way of attaining the benefits of large-scale multiprocessing without surrendering programmability [1]–[6]. An interesting subclass of these machines is the class that provides hardware cache coherence, which makes programming easier, while reducing storage access latencies by caching shared data. While these machines can do well on problems with low levels of data sharing, it is unclear how well caches will perform when accesses to shared data occur frequently.

The cache performance of shared data is the subject of intense ongoing research. Agarwal and Gupta [7] studied locality issues in traces of memory references from a four-processor machine and reported a high degree of processor interleaving in the accesses to a given shared-memory location. This suggests that shared data can be the source of frequent misses. Indeed, Eggers and Katz [8], in a simulation of 5 to 12

processors in a bus, showed that shared data is responsible for the majority of cache misses and bus cycles. In addition, they show that the miss rate of the data cache in multiprocessors changes less predictably than in uniprocessors with increasing cache block size. While the miss rate in uniprocessors tends to go down with increasing cache block size, the miss rate in multiprocessors can go down or up with larger block sizes. A further understanding of the patterns of data sharing was provided by Weber and Gupta [9], who showed that write-shared variables are usually invalidated from caches before being replicated in more than a few different caches. Finally, in another example of unusual behavior, Lee *et al.* [10] found that the optimal cache block size for data is one or two words long, in contrast to the larger sizes used in uniprocessors [11]. Clearly, given the performance impact of the cache behavior of shared data, a deeper understanding of it is necessary.

In this paper, we focus on one parameter that has a major effect on the cache performance of shared data, namely the size of the cache blocks. A second issue that motivates the interest in this topic is that the measurements obtained so far on the impact of the block size on the miss rate of shared data show such wide variation [8] that they are difficult to generalize. In this paper, we explain the effect of the cache block size on the miss rate as a combination of two well-behaved components: false sharing and spatial locality. False sharing, in its simplest form, occurs when two processors repeatedly write to two different words of the same cache block in an interleaved fashion. This causes the cache block to bounce back and forth between the two caches as if the contents of the block were truly being shared. False sharing usually increases with the block size and tends to drive miss rates up with increasing block size. The second component, spatial locality in the data [12], is the property that indicates that the probability of an access to a given memory word is high if neighboring words have been recently accessed. This well-known property produces the opposite effect from false sharing—a reduction in the miss rate as the block size increases.

We assess the contribution of each component by using a model of sharing where individual misses are classified as false sharing misses or as true sharing misses. The latter are due to the interprocessor communication intrinsic to the application. False sharing misses measure false sharing. The effectiveness of increasing the cache block size in eliminating true sharing misses measures the degree of spatial locality present. Experimental measurements show that poor spatial locality in shared data has a larger effect than false sharing in determining the overall miss rate.

Manuscript received June 1990; revised June 1991 and October 1992. This work was supported in part by DARPA Contract N00014-87-K-0828 and by financial support from La Caixa d'Estalvis per a la Vellesa i de Pensions and the Ministerio de Educación y Ciencia, both of Spain.

J. Torrellas is with the Center for Supercomputing Research and Development and Computer Science Department, University of Illinois at Urbana-Champaign, Urbana, IL 61801 USA.

M. S. Lam and J. L. Hennessy are with the Computer Systems Laboratory, Stanford University, CA 94305 USA.

IEEE Log Number 9400784.

TABLE I  
APPLICATION SET CHARACTERISTICS

Application	Description	Shared Data Space (Mbytes)
Csim	Chemical-Mixing logic gate simulator.	3.83
DWF	Performs string pattern matching.	2.10
Ngpt	3-D particle simulator used in aerodynamics.	1.85
LocusRoute	Global router for VLSI standard cells.	1.84
Maxflow	Determines the maximum flow in a directed graph.	0.36
Mincut	Partitions a graph using simulated annealing.	0.01

The third column lists the size of the data structures declared shared.

To reduce the number of cache misses due to poor spatial locality and false sharing, we propose optimizations that require no programmer help and can therefore be implemented by the compiler. Further, we do not consider techniques that require changes to the assignment of computation to processors, as in loop interchange or loop tiling [13], [14], since they are only feasible in highly regular codes. Instead, we propose simple, local techniques that optimize the layout of shared data at the cache block level. These techniques are effective enough to eliminate, on average, about 10% of the misses on shared data in the applications.

This paper is organized as follows. Section II discusses the methodology and characteristics of the application set used throughout the study. Section III presents a model of data sharing. This model is used in Section IV to analyze experimental data on cache miss rates and processor-memory traffic. Based on this analysis, we propose and evaluate optimizations to improve data caching in Section V. Finally, in Section VI, detailed simulations on an existing architecture examine the performance impact of the issues raised in the previous sections.

## II. METHODOLOGY AND APPLICATION SET CHARACTERISTICS

The results reported in this paper are based on simulations driven by traces of parallel applications. The applications are compiled with a conventional optimizing compiler. This section describes the characteristics of the applications used, presents the simulator models, and evaluates the effect of conventional code optimizations on the frequency of data sharing.

### A. Application Set and Trace Characteristics

The parallel applications studied represent a variety of engineering algorithms [15]–[20] (Table I). Csim, Mp3d, and LocusRoute are research tools with between 1000 and 6000 lines of code. The other three applications, namely DWF, Maxflow, and Mincut implement several commonly used parallel algorithms and are less than 1000 lines of code each. Each application uses the synchronization and sharing primitives provided by the Argonne National Laboratory macro package [21]. The synchronization primitives are locks, barriers, and distributed loop control variables. The applications are in C and written so that they can run on any number of processors. We use code compiled with standard code optimization.

We trace the applications using Tango [22], a tracing program that simulates a multiprocessor. The traces correspond to

16 and 32 processor runs of the applications. They contain only application virtual address references and range in size from 8 to over 32 million *data* references. Synchronization variables do not use spin-locking and, to minimize the possibility of hot spots, each synchronization variable is allocated to its own cache block.

### B. Simulated Architectures

Two simulated architectures are used in this paper, the ideal and the detailed architecture. In the ideal architecture, caches are infinite; all memory references, read or writes, hits or misses, take a single cycle; and every instruction executes in one cycle. We use the ideal architecture to remove dependencies on specific architecture characteristics from our study of shared data.

The detailed architecture, used to determine the practical implications of the ideal study, resembles the Silicon Graphics POWER Station 4D/240 [23] in memory system bandwidth and latency. Unlike the 4D/240 system, however, the detailed architecture has 16 processors, each of which has one 256 Kbyte direct-mapped data cache. In addition, synchronization accesses use the same bus as regular transactions. The memory access times without contention for 4- and 16-word blocks are 22 and 31 cycles respectively, during which the bus is locked for 6 and 15 cycles respectively. To simulate a steady state, the applications are executed twice; the first run warms up the cache, and the measurements are taken in the second run. Because bus contention would be too high with 32 processors, the detailed architecture is used for 16 processor runs only.

Both architectures use the invalidation-based Illinois cache coherence protocol [24]. Because in the 4D/240 a request for ownership on a shared block has the same timing and traffic requirements as a cache miss, we do not distinguish between the two in this paper.

### C. Effect of an Optimizing Compiler on the Frequency of Sharing

While code optimizations are known to speed up uniprocessor applications [25], they have an important second effect in multiprocessor code: they increase the frequency of shared data references. This results from the different ways in which optimizations affect data. While some private references are eliminated by register allocation and other optimizations, shared data consistency prevents existing compilers from optimizing data declared shared, even if not used as such. Consequently, since some cycles are saved while the number of shared references remains the same, data sharing has a larger impact on the speed of the application.

To study the effect of an optimizing compiler, we measure, before and after compiler optimization, the fraction of references to data declared shared. The target architecture is the MIPS R2000 processor [26], which has 32 integer registers and 16 double-precision floating point registers. The optimizations applied include global register allocation and other conventional global optimizations. All data in the shared space is declared volatile, and therefore are not register-allocated or optimized. Because optimizations affect the different types of

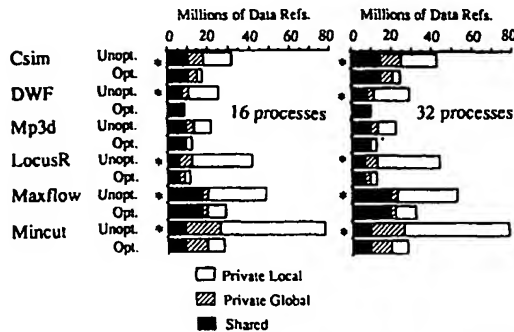


Fig. 1. Decomposition of the optimized and unoptimized data reference streams for 16 and 32 processes.

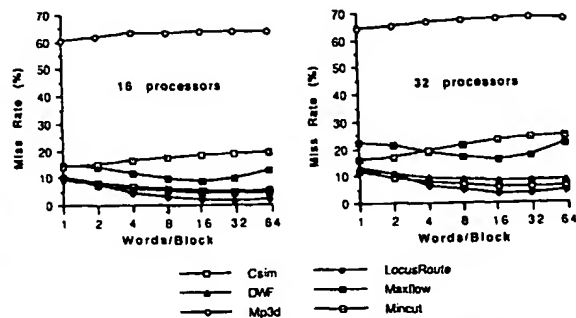


Fig. 2. Cache miss rates on shared data as a function of the block size for the ideal architecture. For a given application, the same problem size is used in the 16- and 32-processor executions.

private data differently, we consider local and global private data separately. *Local* data are the variables declared within procedures. *Global* data is mostly static data set up by the master process for the slave processes.

Fig. 1 shows the decomposition of the data reference streams for the optimized and unoptimized applications running with 16 and 32 processes. Due to limited disk space, the unoptimized versions of some traces were not run to completion (bars with a star). In those cases, the total number of references is calculated assuming the same relative ratios of private local, global, and shared references that existed when the trace was interrupted and the same number of shared references as the optimized trace. From the figure, we see that, for all applications, a large number of private references are eliminated, particularly among those directed to local variables. References to private global variables show a smaller change, almost solely due to the register allocation of the global pointer to the shared data space. shared data references, therefore, account for a larger fraction of references. We also see that the number of processes has little effect on the results. Appendix A shows tables with the actual numbers obtained in the experiments. The large difference in the ratio of shared to total references between optimized and unoptimized code suggests that performance studies of multiprocessor programs must be based on optimized code.

### III. ANALYZING SHARING

Data miss rates in large uniprocessor caches tend to vary predictably as cache blocks increase in size [11], [27], [28]. Initially, the miss rate drops quickly as the block size increases; for large blocks, around 32 words, the curve flattens out; eventually, there is a slight reversal of the curve because of misses resulting from conflicts. In contrast, how miss rates on shared data change with block size is much less predictable; experimental data shows a significant variation across programs (Fig. 2). In this section, we first present a model of sharing that decomposes the widely varying miss rates on shared data in an invalidation-based cache coherence protocol into two well-behaved and intuitive components. Then, we describe an experiment to quantify each of these

Private Data		Shared Data	
Single-Word Cache Blocks	Multi-Word Cache Blocks	Single-Word Cache Blocks	Multi-Word Cache Blocks
Cold Start	Cold Start	Cold Start	Cold Start
	Prefetching	True Sharing	Prefetching
			True Sharing
			False Sharing

Fig. 3. Factors that determine the data misses in an infinite cache.

components. For simplicity, all the analysis in Section III assumes an infinite cache.

#### A. A Model of Sharing

Fig. 3 shows the factors that determine the number of data misses in an infinite cache. For private data in single-word cache blocks, misses are solely caused by first-time references to the data. This effect we call *cold start* in Fig. 3. If the cache has multi-word blocks, the prefetching provided by the multiple words of the block reduces the number of misses, as one miss is enough to bring all the words of a block into the cache. There are several more factors involved with the misses on shared data. If single-word blocks are used, *true sharing* as well as cold start dictate the misses. True sharing is the sharing of the same memory word by different processors. True sharing is intrinsic to a particular memory reference stream of a program and is not dependent on the block size. The presence of multi-word blocks further adds *false sharing* to true sharing, cold start, and prefetching effects. False sharing occurs when different processors access different words of the same block and the coherence protocol forces the block to bounce among caches as if its words were truly being shared. A result of the collocation of different data in the same cache block, false sharing depends on the block size and the particular placement of data in memory. In the following paragraphs, we show how each individual cache miss can be traced back to these factors.

True and false sharing are illustrated in Fig. 4(a), where words *a* and *b* are in the same memory block and an asterisk marks a cache miss. In Examples I, II, and III, processor *P* owns that block at the beginning of the reference stream, since

	Initial State	References	
EXAMPLE I	Pa Pb	Qb* Pa* Qb*	
EXAMPLE II	Pa Pb	Qa* Pb* Pa	
EXAMPLE III	Pa Pb	Qa* Pa* Pb	
(a)			
	Initial State	References	
EXAMPLE I	Pa Pb	Qb* Pa* Qb*	
Block owned by	P P	Q P Q	
a: Reference stream	P	P	
Sharing transitions		F	+1
Successful Prefetches			-0
b: Reference stream	P	Q Q	
Sharing transitions		T F	+2
Successful Prefetches			-0
			misses = 3
EXAMPLE II	Pa Pb	Qa* Pb* Pa	
Block owned by	P P	Q P P	
a: Reference stream	P	Q P	
Sharing transitions		T T	+2
Successful Prefetches		X	-1
b: Reference stream	P	P	
Sharing transitions		F	+1
Successful Prefetches			-0
			misses = 2
EXAMPLE III	Pa Pb	Qa* Pa* Pb	
Block owned by	P P	Q P P	
a: Reference stream	P	Q P	
Sharing transitions		T T	+2
Successful Prefetches			-0
b: Reference stream	P	P	
Sharing transitions		F	+1
Successful Prefetches		X	-1
			misses = 2

(b)

Fig. 4. Example of memory reference streams. For simplicity, the streams contain only writes. An asterisk marks a cache miss. The streams in part (a) are expanded in part (b) showing true (T) and false (F) sharing transitions, and misses saved by successful prefetches (X). Words *a* and *b* share the same cache block. *Pa* means processor *P* writes word *a*.

*P* previously wrote words *a* and *b*—as denoted by *Pa* and *Pb* under "Initial State." In Example I, processor *Q* writes to word *b* and processor *P* writes to word *a*. In this classical case of false sharing, this pattern of access produces a miss for every access. Except for the first *Qb* reference however, no true data sharing is involved. In Examples II and III, processor *P* and *Q* need, and therefore truly share, word *a*. Word *b* is used only by *P* in both cases. However, because of the prefetch provided by the cache block, this common sharing pattern produces misses on different words in the two examples. A more complex sharing pattern can interact with the cache block in a variety of ways, resulting in different number of misses. The model we present now analyzes how data sharing and prefetching interact to result in the observed number of misses.

We assume a multiprocessor with infinite caches and an validation-based cache coherence protocol where a cached memory word may be owned by one cache or read shared

among several. We define the *state* of the word as the pair (*mode*, *processors*), where *mode* may be *owned* or *shared*, and *processors* is the set of processors that cache the word. An uncached word is a degenerate case where *processors* is  $\emptyset$ . A read miss loads the word in a shared mode. If the word is in a shared mode, a processor that caches it must issue a request for ownership before it can write the word. We count this request as a miss. A change in the state of the word is called a *state transition*. In the following, we focus on conditions after the cold start for the word, when no processors will access the word for the processor's first time.

To quantify the degree of intrinsic sharing of a memory word, we define the concept of true sharing transition.

**Definition 1) True Sharing Transition:** Consider the stream *S* of references to a given memory word only and ignore any effects caused by references (not in *S*) to the other words in the same cache block, as if the block were single-worded. We call true sharing transition any state transition that occurs between two references that are contiguous in *S*, after cold start. Further, we say that the second reference causes a true sharing transition.

Example II in Fig. 4(a) shows two true sharing transitions for word *a*. One occurs between the initial state and reference *Qa*; the second between *Qa* and the last reference.

True sharing transitions and cache misses are strongly related: in caches with single-word blocks, every true sharing transition causes a cache miss, and every miss after cold start is due to a true sharing transition. In caches with multi-word blocks, however, a true sharing transition does not necessarily lead to a miss. This is shown in the second true sharing transition for word *a* in the same example. Between the two references involved in the transition, namely *Qa* and *Pa*, a third reference *Pb* to another word of the same block prefetches the original word to the desired state, owned by processor *P*. As a result, the second reference *Pa* hits. On the other hand, the first true sharing transition for word *a* in the same example, which occurs between the initial state and *Qa* produces a miss. We can now define the concept of true sharing miss.

**Definition 2) True Sharing Miss:** A miss that occurs in a true sharing transition.

The previous discussion shows that prefetching can eliminate a miss in the second reference of a true sharing transition. Prefetching can also generate a miss in a reference that does not cause a true sharing transition. To formalize this situation, we first define the concept of false sharing transition.

**Definition 3) False Sharing Transition:** Consider two consecutive references to the same word where the second reference does not cause a true sharing transition. If, between the two references, there is at least one intervening reference to a different word of the same block that induces a transition on the second reference, we say that the second reference causes a false sharing transition.

As an example, the second *Pb* reference in Example II causes the only false sharing transition for word *b* in the stream: between the two *Pb* references, the intervening *Qa* reference changes the state of word *b* to be owned by *Q*, thereby inducing a transition on the second *Pb* reference.

Like a true sharing transition, a false sharing transition may or may not incur a miss. An example where a miss occurs is the false sharing transition for word *b* in Example II: between the two *Pb* references, reference *Qa* leaves the block in state owned by *Q*, causing the second *Pb* reference to miss. An example where the miss is avoided is shown in Example III, which is equal to Example II with the last two references flipped. In Example III, the second *Pb* reference causes a false sharing transition because reference *Qa* between the two *Pb* references induces a transition on the second *Pb*. Between *Qa* and *Pb*, however, reference *Pa* brings the lock back to *P*'s ownership, thus successfully eliminating a cache miss in the *Pb* reference. We can now define the concept of false sharing miss.

**Definition 4) False Sharing Miss:** A miss that occurs in a false sharing transition.

Finally, based on the above definitions, the total number of cache misses, not counting the cold start effect, is the total number of true and false sharing transitions minus the number of successful prefetches. This equality is illustrated in Fig. 4(b), which expands the streams in Fig. 4(a). We analyze Example II carefully here; the reader is encouraged to go over the other examples. We consider the first reference after the initial state *Qa* and ask whether it causes a false sharing transition (FST), a true sharing transition (TST), or no transition at all. To answer this question, we look at the previous reference to the same word, namely *Pa*. We note that the two references are involved in a TST. We then check whether the accesses between the two references leave word *a* in the state that *Qa* requires, namely owned by *Q*. If that were the case, a successful prefetch would be recorded. Otherwise, the actual situation in the example, a true sharing miss (TSM) occurs. We now consider the next reference *Pb*. As before, we look for the previous reference to the same word, namely the *Pb* under Initial State. This pair of references are not involved in a TST. To check whether a FST occurs, we search the intervening references for at least one that induces a transition on the second *Pb*. Since *Qa* induces such a transition, *Pb* causes a FST. To determine whether a false sharing miss (FSM) or a successful prefetch occurs, we check whether the stream between *Qa* and *Pb* leaves *b* in the state required by *Pb*. Since this is not the case, a FSM occurs. The final reference causes a TST but a successful prefetch eliminates the miss: reference *Pb* sets the block to the desired state, namely owned by *P*. To summarize, the net result of three transitions and one successful prefetch is two misses, as postulated by our equality that relates misses, transitions, and successful prefetches.

#### B. Effect of Data Prefetching Through Increased Block Size

The previous analysis showed that the prefetching provided by multiword blocks can eliminate or create misses. Unlike in uniprocessors, where prefetching always has a positive effect in infinite caches, prefetching in multiprocessors can have both a positive and a negative effect. Prefetching exploits spatial locality in data as in uniprocessors. It also, however, creates false sharing transitions, which may change what used to be cache hits without prefetching into false sharing misses.

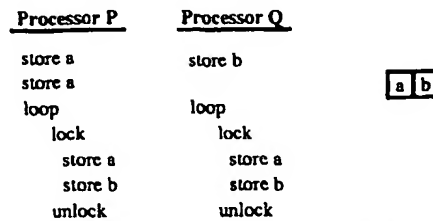


Fig. 5. The data prefetching provided by multi-word cache blocks can be beneficial, as in the loop shown, or may create false sharing misses, as in the statements before the loop.

We expect the positive effect, namely exploitation of spatial locality, to be lower in multiprocessors than in uniprocessors for three reasons. First, a processor may never reference the prefetched data: since computation is partitioned in a multiprocessor, this is more likely than in a uniprocessor. Second, even if the processor will eventually access the prefetched data, another processor may access it first and remove the data from the first processor's cache. Third, prefetched data may be removed by another processor accessing a different word in the same block. Because of the last reason, the benefits of spatial locality do not necessarily increase monotonically with the cache block size. Larger blocks may introduce transitions that reduce the spatial locality benefits present in a smaller block size.

False sharing transitions, the second effect of prefetching, increase monotonically with block size. As false sharing transitions increase, false sharing misses are likely to increase. However, since not every false sharing transition will cause a false sharing miss, the number of false sharing misses may not increase monotonically with increasing block size either.

Unfortunately, both the positive and negative effects of prefetching are determined by the particular placement of data in memory and cannot, in general, be changed independently. Fig. 5 illustrates the interdependence of the two effects. In the figure, words *a* and *b* share the same block. In the beginning of the program, a potential instance of false sharing occurs because processor *Q* may write *b* while processor *P* writes *a*. During the rest of the program, the two processors access words *a* and *b* in sequence within a critical section. If we eliminated the false sharing by, for example, placing *a* in a different block, the benefits of prefetching within the loop would also disappear. We could be saving one false sharing miss at the cost of doubling the number of misses within the loop. This example suggests that it may not be desirable to eliminate false sharing misses at any cost.

#### C. Measurements

Although the positive and the negative effects of prefetching on the miss rate are closely related, we have been able to devise an experiment that allows us to measure each of the two effects. The experiment is based on our model of sharing. In the experiment, we use the ideal architecture, which assumes infinite caches and no cache miss penalties. We compare two simulations driven by the same interleaving of references and running in lockstep. One simulation uses caches with single-

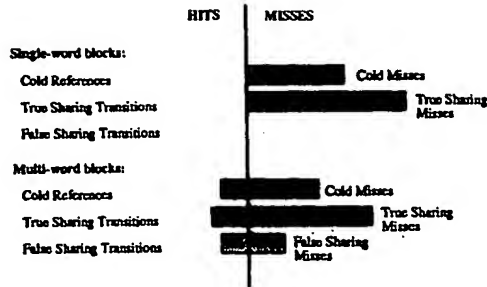


Fig. 6. Relation between the simulation of the ideal architecture using single-word and multiword cache blocks. The figure is not drawn to scale. In the figure, the number of cold references and true sharing transitions are the same in both simulations.

word blocks, while the other uses caches with multiword blocks. In the simulations, we include the cold start period of the programs. Hence, in addition to false and true sharing misses, we capture misses on memory words referenced by a processor for the processor's first time. These misses we call *cold misses*. The relationships among cold, true sharing, and false sharing misses in the single-word and the multiword simulations are as follows.

- If a cold miss is incurred for a reference in the multiword block simulation, the same reference causes a cold miss in the single-word block simulation.
- True sharing transitions are intrinsic to a reference stream of a program and thus identical for both simulations. Since all true sharing transitions result in misses (true sharing misses) in the single-word case, if a true sharing transition in the multiword block simulation causes a miss, it also causes a miss in the single-word block simulation.
- Therefore, the remaining misses, those that occur in the multiword block simulation but not in the single-word case, must be all false sharing misses.

In summary, comparing the two simulations, a miss in the multiword simulation is a false sharing miss if there is no equivalent miss in the single-word case; otherwise it is a cold or true sharing miss.

Fig. 6 (not drawn to scale) depicts the relationships described. The number of cold references and true sharing transitions are the same in both simulations. Prefetching multiple words in a cache block has two effects: first, some of the cold references and true sharing transitions now result in hits; second, false sharing transitions appear, some of which result in hits and some in misses.

#### IV. ANALYZING THE CACHE MISS RATE AND TRAFFIC BEHAVIOR OF SHARED DATA

In this section, we use the experiment just described to analyze the cache miss rates and traffic generated by shared data in real applications. To eliminate dependencies on specific architecture characteristics, we use the ideal architecture throughout the section. We start with an analysis of the miss rates; then we consider the traffic behavior.

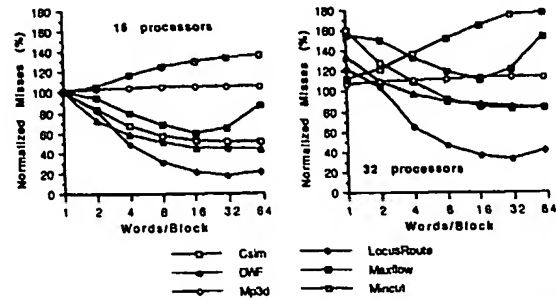


Fig. 7. Cache misses on shared data as function of the block size. Misses are shown as a fraction of the misses on single-word blocks for the same application and 16 processors.

#### A. Analysis of the Cache Miss Rates on Shared Data

Fig. 2 shows the miss rates on shared data as a function of the block size for the applications studied. We observe a wide variation among applications, both in absolute values and in the way the block size affects them. For example, whereas miss rates for Csim, DWF, and LocusRoute start from relatively low values and decrease with increasing block size, Maxflow's miss rate starts with a higher value, decreases at first, and then increases. Mp3d's miss rate is high and not very sensitive to changes in the block size. Finally, Mincut shows an upward trend.

To understand the variation observed with changes in the block size, we plot the miss curves in relative values (Fig. 7) and then decompose them into the miss components as described by our model. Fig. 8 shows the misses for 16 processors decomposed into two groups: cold and true sharing misses, and false sharing misses. In addition, to show the degree of true sharing in each program, we mark with an arrow the number of true sharing misses on single-word blocks—which is also the number of true sharing transitions. The rest of the misses on single-word blocks are cold misses. In the following sections, we first analyze each component of the misses separately, relating the shape of the curves to the data structures in the program that cause them. Then, we summarize the general observations.

**Analyzing False Sharing Misses:** Recall that, while false sharing transitions always increase monotonically with the block size, this is not necessarily so for false sharing misses. From Fig. 8, however, we observe that false sharing misses always increase with block size and that, except in two cases, this increase is slow. This slow increase is produced by several program characteristics. Distributing the computation such that each iteration of a loop is executed on a different processor produces false sharing misses when data from different iterations falls in the same cache block. Graph problems with irregular node interconnection where cache blocks frequently contain pieces of nodes belonging to different processors also exhibit false sharing misses (Maxflow and Mincut).

The two cases where false sharing misses increase quickly are when the blocks are small in Mincut and when the blocks are large in Maxflow. The sharp rise in these two cases is due to the presence of blocks containing multiple frequently-

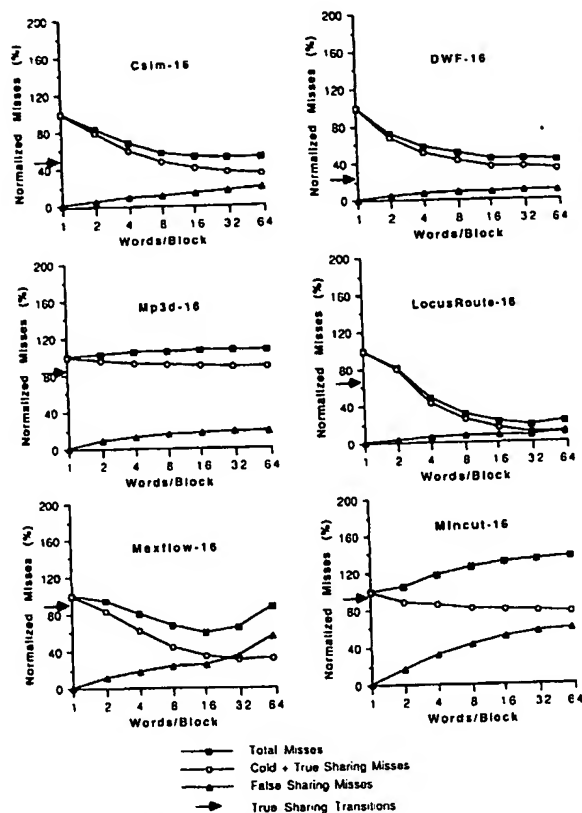


Fig. 8. Decomposition of the cache misses on shared data as a function of the block size for 16 processors. Misses are shown as a fraction of the misses on single-word blocks for the same application. The arrow shows the number of true sharing transitions in the program.

accessed scalar variables, where at least one of the scalars is written frequently. This effect can also happen with small arrays where each array entry is repeatedly updated by one processor.

As opposed to the previous applications, programs with little reuse of data by the same process (Mp3d), or where each processor is assigned a geographic domain where processor interaction is infrequent (LocusRoute, DWF and Csim) are unlikely to exhibit a large amount of false sharing misses.

**Analyzing Cold and True Sharing Misses:** The slow decrease in cold and true sharing misses with increasing block size seen in Fig. 8 shows that shared data has low spatial locality. A second observation is that, except for Maxflow, which shows a slight trend reversal for large blocks, the decrease in misses is monotonic.

Poor locality particularly affects programs with unstructured accesses, as is the case in fine-grained global task queues where processors continually process new tasks (Mp3d) or algorithms like simulated annealing that involve calls to random number generators to decide what memory area to access (Mincut). On the other side, programs with large data structures that are accessed sequentially and at different times by

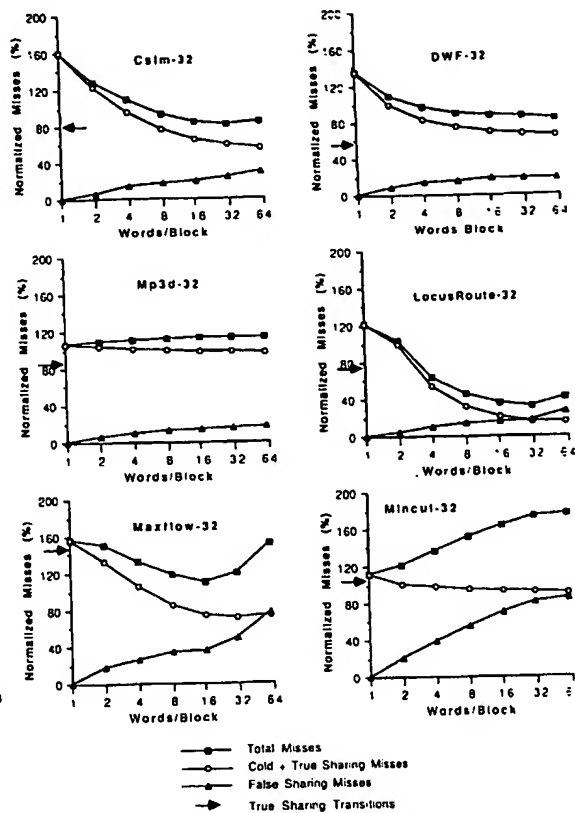


Fig. 9. Decomposition of the cache misses on shared data as a function of the block size for 32 processors. Misses are shown as a fraction of the misses on single-word blocks for the same application and 16 processors. The arrow shows the number of the true sharing transitions in the program.

different processors (LocusRoute and Maxflow) show larger decreases in cold and true sharing misses.

**Increasing the Number of Processors:** The curves with the misses for 32 processors shown in Fig. 7 are decomposed in Fig. 9. The misses are shown as a fraction of the misses on single-word block misses for the same application. From the figure, we see that the two components, false sharing and cold/true sharing misses, maintain the same trends for the larger number of processors.

**Overall Observations:** Despite the widely varying shape of the overall curve, the two component curves behave consistently across all applications. First, cold and true sharing misses tend to decrease with increasing block size but, unlike in uniprocessors, the rate of decrease in misses is much less than the rate of increase in block size. Second, false sharing misses increase with block size and eventually neutralize or even overcome the small decreases in the cold and true sharing misses. The net effect is that the total number of misses either decreases slowly or does not decrease at all. As we will see, the result is a dramatic increase in processor-memory traffic with any increase in block size.

The plots show that cold and true sharing misses usually outnumber false sharing misses. Further, for the two applications

with a significant number of false sharing misses, we show in Section V that simple data placement optimizations can eliminate an important fraction of these false sharing misses.

The two component curves in each plot may not be independent of each other. Fig. 5 showed that a reduction in the number of false sharing misses may cause an increase in the number of cold and true sharing misses. The opposite case, namely a reduction in the number of false sharing misses causing a decrease in the amount of cold and true sharing misses, is also possible. Such scenario occurs if false sharing misses induce more misses by interfering with the successful prefetches for true sharing or cold accesses. In the worst case, a false sharing miss on a word by one processor could eliminate a successful prefetch in all the other processors that cache the word, thereby forcing cold or true sharing misses. Fortunately, the experiments performed while studying the optimization of Section V show that such interaction is rare. The curves of false sharing misses, therefore, are a good approximation of the worst effects of false sharing.

The magnitude of the two component curves and the previous discussion suggest that the poor spatial locality of multiprocessor data—responsible for the slow decrease in cold and true sharing misses—contributes to the cache miss rates even more than false sharing does. For this reason, we believe that, to improve the performance of caches, trying to enhance the spatial locality of multiprocessor data is an approach at least as, or even more promising, than trying to remove false sharing.

#### B. Analysis of the Traffic Generated by Shared Data

Not only do misses increase the latency of memory accesses, they also generate traffic between processors and memory. As the block size increases, a miss produces a higher volume of traffic. If we estimate the traffic caused by shared data as  $\text{SharedMisses} \times \text{BlockSize}$ , we produce the plots in Fig. 10. The figure includes a curve for uniprocessor data with a finite cache (32 Kbytes) from [11] for comparison purposes. From the figure, we see that the block size that minimizes the traffic of shared data in this class of applications is one word, both for 16 and 32 processors. The highest performance block size, however, is larger than that. Indeed, to determine the highest performance block size for a data cache, we need to take into account the start up overhead associated with a cache miss for the particular machine and know what fraction of the data misses are on shared data. Section VI shows that this fraction is over 95% for a large cache.

The traffic increase with larger blocks occurs because many of the words transferred are not used. Between two consecutive misses on a given block, a processor usually references a very small number of distinct words in that block, as shown in Table II. Recall that misses include requests for ownership on a block. The low values in Table II show that, on average, the prefetching effect of cache blocks is not very effective. These numbers correlate with the trends in the miss rates shown in Fig. 2. Mp3d has the lowest numbers in Table II because it has a high miss rate, which does not decrease with larger block sizes. LocusRoute shows the highest numbers

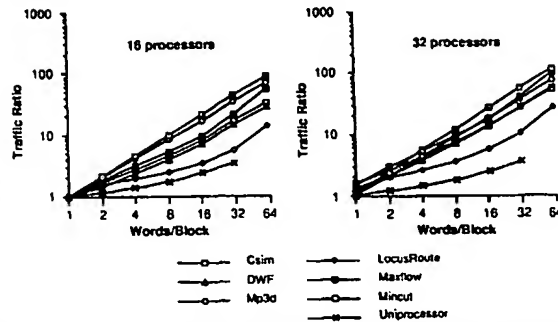


Fig. 10. Processor-memory traffic caused by shared data. The plot shows the ratio between the traffic at a given block size and the traffic for single-word blocks and 16 processors. We include a curve for uniprocessor data with a finite cache (32 Kbytes) for comparison purposes.

TABLE II  
AVERAGE NUMBER OF DISTINCT WORDS IN A CACHE  
BLOCK REFERENCED BY ONE PROCESSOR BETWEEN TWO  
CONSECUTIVE MISSES ON THAT BLOCK BY THE SAME PROCESSOR

Application	16 Processors					32 Processors				
	Block Size (Words)					Block Size (Words)				
	2	4	8	16	32	2	4	8	16	32
Csim	1.4	1.9	2.4	3.0	3.5	1.4	1.8	2.3	2.7	3.2
DWF	1.5	1.9	2.4	2.3	2.5	1.4	1.7	1.9	2.0	2.1
Mp3d	1.1	1.1	1.1	1.1	1.1	1.0	1.1	1.1	1.1	1.1
LocusRoute	1.3	2.1	3.5	5.2	6.0	1.2	2.0	3.0	4.1	4.3
Maxflow	1.3	1.9	2.5	3.1	3.3	1.2	1.7	2.0	2.3	2.5
Mincut	1.2	1.2	1.4	1.7	2.0	1.1	1.2	1.3	1.4	1.7

because it has a low miss rate that decreases significantly with increases in block size. We also see that increasing the number of processors always decreases the number of words used in a block. The poor use of the cache blocks revealed by this data motivates the next section, where we try to optimize the use of the blocks based on our model of sharing.

#### V. OPTIMIZING THE PLACEMENT OF SHARED DATA IN CACHE BLOCKS

This section addresses the problem of reducing the cache misses on shared data by enhancing the spatial locality of shared data and mitigating false sharing. We optimize the placement of data structures in cache blocks using local changes that are programmer-transparent and have general applicability. Our approach is partly motivated by the fact that cache misses on shared data are often concentrated in small sections of the shared data address space. Therefore, local actions involving relatively few bytes may yield most of the desired effects. An example of this skewed miss distribution is shown in Fig. 11, which plots the average number of misses per byte in each shared data structure of Csim.

To guide the study of possible optimization, we use address traces to generate the following profiling information for each shared-memory word: 1) degree of true sharing, measured as the number of misses beyond the cold start in the single-word block simulation, 2) false sharing misses, 3) cold and true

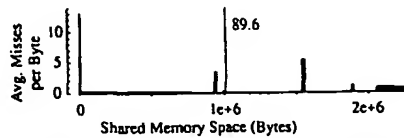


Fig. 11. Distribution of the cache misses along the shared address space for the Csim application. For each data structure, we plot the average number of misses per byte. This plot corresponds to 16 processors, 4-word cache blocks, and the ideal architecture.

sharing misses eliminated by prefetching, and 4) number of writes. The latter is needed since, in addition to words that have a high degree of true sharing, non-shared words that are frequently written can also be the cause of false sharing in a block. For example, false sharing may occur in a block with one word that is heavily read by only one processor and one word that is heavily written by only one other processor. We call a word *active* if its degree of true sharing or number of writes exceeds 0.1% of the program misses.

In the following, we first present the optimizations, then evaluate them using the ideal architecture. In the evaluation, we consider both the aggregate effect of all optimizations and the individual effect of each. Since it is rare to have this tracing information in practice, the final subsection examines the case where we have no dynamic information on the application at all.

#### A. Placement Optimizations

We propose five optimizations of the data layout. Because synchronization variables are a well-known source of contention in some programs, we use as a baseline a data layout where each of them is allocated to an empty cache block.

- *SplitScalar*: Place scalar variables that cause false sharing in different blocks. Given a cache block with scalar variables where the increase in misses due to prefetching exceeds 0.5% of the program misses, we remove the active variables and allocate each of them to an empty cache block.
- *HeapAllocate*: Allocate shared space from different heap regions according to which processor requests the space. It is common for a slave process to access the shared space that it requests itself. If no action is taken, the space allocated by different processes may share the same cache block and lead to false sharing. The policy we propose is more space-efficient than allocating only block-aligned space, particularly when very small chunks of space are repeatedly requested.
- *Expand Record*: Expand records in an array (padding with dummy words) to reduce the sharing of a cache block by different records. While successful prefetching may occur within a record or across records, false sharing usually occurs across records, when more than one of them share the same cache block. If the multi-word simulation indicates that there is much false sharing and little gain in prefetching, then consider expansion. If the reverse is true, do not apply the optimization. When both false sharing misses and prefetching savings are of the same order

of magnitude, we assume that the prefetching succeeds within a record and we apply the optimization.

- *Align Record*: Choose a layout for arrays of records that minimizes the number of blocks the average record spans. This optimization maximizes prefetching of the rest of the record when one word of a record is accessed, and may also reduce false sharing. This optimization is possible when the number of words in the record and in the cache block have a greater common divisor (GCD) larger than 1. The array is laid out at a distance from a block boundary equal to 0 or a multiple of the GCD, whichever wastes less space.
- *LockScalar*: Place active scalars that are protected by a lock in the same block as the lock variable. As a result, the scalar is prefetched when the lock is accessed.

All optimizations except *LockScalar* try to minimize false sharing. *LockScalar* and *AlignRecord* try to increase the spatial locality of the data. In our optimization, we must avoid other effects that could offset the intended ones. First, false sharing and effective exploitation of spatial locality are not independent; changing one usually affects the other. In particular, strategies that increase the size of the data like *SplitScalar* and *ExpandRecord* may also reduce the effectiveness of prefetching in eliminating cold and true sharing misses. Second, large data expansions may increase the working set of a program and increase capacity misses in a finite cache. To guard against these effects, we restrict the optimizations to those that cause little data size increases.

#### B. Evaluation of the Optimizations: Aggregate Effect

To evaluate the effectiveness of these optimizations, we use as a metric the fraction of shared data misses that they eliminate. Table III shows this fraction together with the resulting increase in the size of the data structures for 16 and 32 processors with 4- and 16-word blocks. The table shows a large variation in the fraction of misses eliminated in the different applications: the results for individual programs range from 0% to over 40%, with an average close to 10%.

On average, our techniques tend to eliminate a higher percentage of misses for the larger block sizes. This effect is, however, the result of two opposing trends. On one hand, a larger cache block size increases the possibility of false sharing among scalars and small data structures, thus possibly increasing the effectiveness of the optimizations. On the other hand, a larger block also increases the cost of expanding records, making some data expansion optimizations infeasible. Further, a larger block may already benefit more from prefetching, rendering optimizations to increase spatial locality less effective.

The effect of the number of processors is also clear. When the number of processors increases, there are more cache misses. The data placement optimization, however, also eliminates more misses. The result, for nearly all the applications studied, is that the relative miss reductions are higher for 32 processors than for 16 processors.

Finally, we see that the space requirements of the optimization are small, usually in the 2 Kbyte neighborhood. This

TABLE III  
EFFECTIVENESS OF THE OPTIMIZATIONS IN THE PLACEMENT  
OF SHARED DATA FOR 16 (TOP HALF OF THE TABLE)  
AND 32 (BOTTOM HALF OF THE TABLE) PROCESSORS

Application	Number of Processors	Reduction in Shared Data Misses				Increase in Shared Data Space			
		4-Word Blocks		16-Word Blocks		4-Word Blocks		16-Word Blocks	
		Relat. (%)	Absol. (Thous.)	Relat. (%)	Absol. (Thous.)	Relat. (%)	Absol. (Kbytes)	Relat. (%)	Absol. (Kbytes)
Csim	16	7.9	60.6	6.6	39.3	0.0	0.4	0.1	1.9
DWP	16	0.6	3.1	1.0	4.6	0.0	0.0	0.0	0.2
Mp3d	16	0.4	20.6	0.1	5.5	0.3	4.8	0.0	0.5
LocusRoute	16	10.2	45.3	28.7	57.5	0.0	0.5	0.1	1.6
Maxflow	16	8.9	198.9	14.2	235.3	0.6	1.6	0.6	1.6
Mincut	16	19.7	229.6	8.9	153.1	72.3	4.0	0.0	0.0
AVERAGE	16	8.0		9.9			1.9	0.1	1.0
Csim	32	15.6	196.2	11.5	110.3	0.0	0.9	0.1	3.9
DWP	32	0.6	5.5	1.1	9.0	0.0	0.0	0.0	0.2
Mp3d	32	0.4	24.3	0.2	13.3	0.3	4.8	0.0	0.5
LocusRoute	32	15.5	92.5	41.6	138.5	0.0	0.5	0.2	3.1
Maxflow	32	10.7	397.0	14.7	455.6	0.6	1.6	0.6	1.6
Mincut	32	22.0	394.1	8.8	190.9	72.3	4.0	0.0	0.0
AVERAGE	32	10.8		13.0			2.0	0.2	1.5

causes an insignificant relative increase in shared data space unless the size of the shared data space is very small originally. While it is possible to reduce the miss rate further by larger data expansions, their possibly detrimental effect on cache performance makes them undesirable.

Fig. 12 shows how the optimizations affect the two types of misses: cold and true sharing, and false sharing misses. For each application, the figure considers the four processor and block size settings used in the previous table. For each setting, we show three bars. The leftmost bar shows the miss rate of shared data in the original program, where the compiler did not necessarily allocate each synchronization variable to a different cache block. The central bar shows the miss rate after each synchronization variable is allocated to a different cache block. This is the miss rate taken as a baseline. From the difference between the two bars, we can see the importance of the synchronization variable layout, especially considering that spin-locking is not used in the synchronization variables. Finally, the rightmost bar shows the miss rate after further applying the five placement optimizations.

We observe that the optimizations are more successful in eliminating false sharing misses than in eliminating cold and true sharing misses. For all applications, the maximum reduction in cold and true sharing misses is approximately 10%. In contrast, almost all false sharing is removed in LocusRoute and in Mincut for 4-word cache blocks, and 20 to 40% in Csim and Maxflow. The reduction of false sharing in Mincut is accompanied by an increase in cold and true sharing misses. This observation illustrates that, in general, the positive and negative effects of prefetching discussed in Section III-B cannot be totally separated.

### C. Evaluation of the Optimizations: Individual Effect

Table IV shows the contribution of each optimization to the reduction in shared data misses shown in Table III. From Table IV, we see the *SplitScalar* is effective for all applications amenable to these optimizations. Most of the

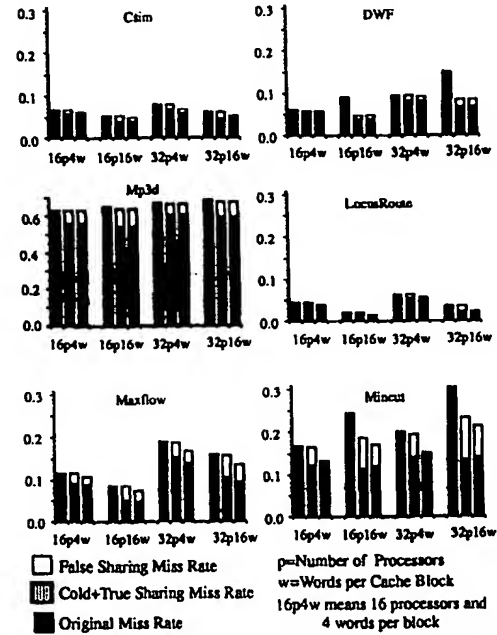


Fig. 12. Miss rates on shared data. For each set of three bars, the leftmost one shows the miss rate of the original program; the central one the miss rate after allocating synchronization variables to different cache blocks; and the rightmost one the miss rate after further applying the five placement optimizations.

TABLE IV  
FRACTION OF SHARED DATA MISSES ELIMINATED BY EACH OPTIMIZATION

Application	SplitScalar	HeapAllocate	ExpandRecord	AlignRecord	LocScalar	Total
Csim 16p-4w	1.7		2.3	3.4	0.5	7.9
Csim 16p-16w	2.2		2.8	1.0	0.6	6.6
Csim 32p-4w	3.8		2.3	9.0	0.5	15.6
Csim 32p-16w	5.0		2.8	3.0	0.7	11.5
DWP 16p-4w	0.4			0.2		0.6
DWP 16p-16w	1.0					1.0
DWP 32p-4w	0.5			0.1		0.6
DWP 32p-16w	1.0			0.1		1.1
Mp3d 16p-4w	0.1		0.3			0.4
Mp3d 16p-16w	0.1					0.1
Mp3d 32p-4w	0.1		0.3			0.4
Mp3d 32p-16w	0.1			0.1		0.2
LocusRoute 16p-4w	1.5	6.7	0.7	0.5	0.8	10.2
LocusRoute 16p-16w	8.0	16.1	2.5	0.4	1.7	28.7
LocusRoute 32p-4w	1.4	11.3	1.4	0.8	0.6	15.5
LocusRoute 32p-16w	8.4	27.5	4.7	0.4	0.6	41.6
Maxflow 16p-4w	1.5		5.3			6.8
Maxflow 16p-16w	2.0		9.3			11.3
Maxflow 32p-4w	1.7		4.8			6.5
Maxflow 32p-16w	2.0		7.7			9.7
Mincut 16p-4w	4.7		9.6		5.4	19.7
Mincut 16p-16w	4.1				4.8	8.9
Mincut 32p-4w	6.7		11.3		4.0	22.0
Mincut 32p-16w	5.5				3.3	8.8

The notation 16p-4w means 16 processor execution and 4 words per cache block.

misses it eliminates can be attributed to two or three actively written scalars. As the block size increases, this optimization becomes more important. We also see the *ExpandRecord* is useful for the same set of applications. This optimization is

applied either to small, active arrays used mainly for process communication or to the main data structures in the smaller programs. Finally, the other optimizations are relevant to only one or two of the applications.

A large fraction of the cache misses still remains after optimization. While some of the false sharing misses can be removed if the data caches are large enough to support more instances of the expansion optimization, the remaining misses are primarily cold and true sharing misses. This suggests that further optimizations should concentrate on increasing the spatial locality of the data.

#### D. Effectiveness of the Optimizations Without Program Profiling

The optimizations evaluated above were developed by using detailed information obtained by tracing the program. While some kind of profiling may be available in practice, it will probably not be as complete as the one used so far. In this section, we investigate the possibility of general and effective optimizations that do not rely on any profiling information. We consider how to apply each of the previous optimizations in the absence of this information:

- *SplitScalar*: If no information is provided, we place *each* shared scalar variable in a different cache block. This approach has almost the same effect as moving only active scalar variables since, in relatively large caches, the advantage of prefetching scalars is minor. Although most programs have a small number of shared scalars (the number in those studied ranged from 5 to 50), programs with many scalars and large cache blocks may waste much space. However, we expect little negative effect, since only a fraction of the scalars is accessed frequently.
- *ExpandRecord*: To expand all short arrays by placing, for example, one entry per cache block is impractical, since it wastes space and can have a positive or a negative net effect on cache misses. We leave it up to the programmer to pad the data structure if so desired.
- *HeapAllocate and AlignRecord*: The optimizations of allocating shared data from a process' own heap space and aligning arrays can be applied at all times, since the cost is low.
- *LockScalar*: If the machine allows lock variables and general data to reside in the same cache block, this optimization is feasible at a very low cost.

From the previous discussion, we conclude that the compiler and run time system can incorporate *Heap Allocate*, *AlignRecord*, *LockScalar*, and the modified *SplitScalar* without any profile information. The cumulative effect of these optimization is shown in Table V, together with a comparison to the fully optimized case. These numbers indicate that a significant part of the effect of the more costly optimizations can be obtained without any profile information. Moreover, the increase in data space, both absolute and relative, remains small.

## VI. PERFORMANCE OF A REAL ARCHITECTURE

After having studied data sharing in an ideal setting, we now use the detailed architecture to illustrate the performance

TABLE V  
EFFECTIVENESS OF THE OPTIMIZATIONS WITHOUT USING A PROGRAM PROFILE

Application	Shared Data Misses Eliminated (%)				Shared Data Space Increase (Kbytes)			
	16 Proc. 4-Word Block	16 Proc. 16-Word Block	32 Proc. 4-Word Block	32 Proc. 16-Word Block	16 Proc. 4-Word Block	16 Proc. 16-Word Block	32 Proc. 4-Word Block	32 Proc. 16-Word Block
Calm	5.6 (71)	3.8 (58)	13.3 (85)	8.7 (76)	0.6	2.9	0.6	2.9
DWF	0.6 (100)	1.0 (100)	0.6 (100)	1.1 (100)	0.1	0.6	0.1	0.6
Mp3d	0.0	0.1	0.0	0.1	0.4	1.9	0.4	1.9
LocusRoute	8.9 (87)	25.9 (90)	14.0 (90)	36.9 (89)	0.6	2.6	0.6	2.6
Maxflow	3.6 (40)	4.9 (35)	5.9 (55)	7.0 (48)	0.1	0.4	0.1	0.4
Miscel	10.1 (51)	8.9 (100)	10.7 (49)	8.8 (100)	0.0	0.3	0.0	0.3
AVERAGE	4.8 (60)	7.4 (75)	7.4 (69)	10.4 (80)	0.3	1.4	0.3	1.4

The numbers in parentheses show the misses eliminated as a percentage of the misses eliminated with a full program trace.

TABLE VI  
DATA CACHE MISS MEASUREMENTS FOR THE DETAILED ARCHITECTURE WITH 16 PROCESSORS AND COMPILER-OPTIMIZED CODE

Application	Private Misses / Private References (%)		Shared Misses / Shared References (%)		Shared Misses / Total Misses (%)		Total Misses / Total References (%)	
	4-Word Block	16-Word Block	4-Word Block	16-Word Block	4-Word Block	16-Word Block	4-Word Block	16-Word Block
Calm	0.1	0.2	9.5	7.7	99	98	5.9	4.8
DWF	0.0	0.1	2.9	2.1	99	99	2.6	1.9
Mp3d	0.2	0.2	59.5	59.3	99	99	46.5	46.3
LocusRoute	0.5	0.3	10.6	5.9	95	94	5.5	3.0
Maxflow	0.3	0.3	13.4	10.0	98	97	8.3	6.2
Miscel	0.0	0.0	19.6	19.1	99	99	6.6	6.4

impact of data sharing in practice. This section examines three issues. We first study the effect of the conventional code optimizations described in Section II-C. Using optimized code, we then measure the overall cache performance of the applications. Finally, also using optimized code, we assess the effectiveness of the placement optimizations for shared data.

#### A. Impact of the Conventional Code Optimizations

To study the effect of the conventional code optimizations on overall performance, we compare the execution times of two applications, LocusRoute and Maxflow, using optimized and unoptimized code. LocusRoute is about twice as fast after optimization for both 4- and 16-words blocks. However, Maxflow yields an improvement of only about 5% for both 4 and 16 word blocks. This small improvement in Maxflow is due to increased bus contention, which offsets the advantages gained by the elimination of unnecessary private data fetches from the program. Thus, while there is a slight improvement in the speed of Maxflow, the utilization of the processors actually decreases by 25%. In conclusion, while some programs run substantially faster with compiler optimizations, those where shared data traffic saturates the interconnection cannot. In either case, since uniprocessor programs run faster while the amount of sharing remains unchanged, optimized code will give lower speedup figures. Since we are ultimately interested in overall performance, measurements on multiprocessor programs must be performed on optimized code.

#### B. Overall Cache Performance

In previous sections, we studied the cache miss rates resulting from data sharing in isolation. In this section we examine

TABLE VII  
EFFECT OF THE SHARED DATA PLACEMENT OPTIMIZATIONS ON  
THE DATA MISS RATES OF THE DETAILED ARCHITECTURE

Application	Overall Data Miss Rate					
	4-Word Block			16-Word Block		
	Unopt. (%)	Opt. (%)	Unopt. - Opt. (%)	Unopt. (%)	Opt. (%)	Unopt. - Opt. (%)
Celm	5.9	5.1	0.8	4.8	4.4	0.4
DWP	2.6	2.6	0.0	1.9	2.0	-0.1
Mp3d	46.5	46.4	0.1	46.3	46.2	0.1
LocustRoms	5.5	4.0	1.5	3.0	1.8	1.2
Maxflow	8.5	7.2	1.1	6.3	5.1	1.1
Miscst	6.6	5.4	1.2	6.4	5.5	0.9

The numbers correspond to 16 processors and compiler-optimized code, and include both shared and private data.

the data cache performance of the detailed architecture, which has a finite cache and issues private data references too. As indicated in Section II-B, the measurements are taken during the steady state execution of the programs. In this environment, the contribution of private and shared data to the misses of the finite caches is shown in Table VI. Because the caches are reasonably large and the programs are measured in their steady state, the miss rate on private data (columns 2 and 3) is minuscule compared to that on shared data (columns 4 and 5). In fact, most of the misses correspond to shared data (columns 6 and 7). Consequently, as shown in the last two columns of Table VI, the total miss rate is basically the shared data miss rate weighted by the frequency of shared data accesses.

### C. Impact of the Placement Optimizations

We have proposed two sets of data placement optimizations: one when full tracing information is available; the other when no profiling data is available. In practice, some information will probably be available. We therefore choose to evaluate the case that assumes full information and consider the results optimistic.

Table VII shows the reduction in data miss rate achieved by the placement optimizations for 16 processors. The data in the table includes misses on both shared and private data. From the table, we see that the optimizations reduce the overall data miss rate of the applications by up to an absolute 1.5% (or a relative 40%). These miss rate reductions speed up the applications by about 10% on average. These speedups are partially the result of the bus contention generated by sixteen processors. However, while replacing the bus with another interconnection network may reduce contention, it may also increase overall memory access latencies.

## VII. CONCLUSION AND FUTURE DIRECTIONS

There are two main contributions in this paper. First, we show how poor spatial locality in the data and false sharing explain the variation in the miss rate of shared data as the cache block changes in size. Second, we show that data layout optimizations that are programmer-transparent and not restricted to regular codes can be used to reduce the miss rate.

Based on the analysis of six applications, we find that, although false sharing sometimes plays a significant role, poor spatial locality has a larger effect in determining the high miss

TABLE VIII  
DECOMPOSITION OF THE DATA REFERENCE STREAM FOR 16 PROCESSES

Application	Shared Refs. / Total Data Refs. (%)		Private Global Refs. / Total Data Refs. (%)		Private Local Refs. / Total Data Refs. (%)	
	Unopt.	Optim.	Unopt.	Optim.	Unopt.	Optim.
Celm	34.1	61.4	22.6	22.6	43.4	16.0
DWP	29.9	91.4	11.7	1.2	58.4	7.5
Mp3d	43.6	78.1	17.2	0.0	39.2	21.9
LocustRoms	13.7	50.2	15.3	22.5	71.0	27.3
Maxflow	36.1	61.1	6.2	6.9	57.7	32.0
Miscst	11.9	33.7	21.5	37.3	66.7	29.1
AVERAGE	28.2	62.6	15.7	15.1	56.1	22.3

TABLE IX  
DECOMPOSITION OF THE DATA REFERENCE STREAM FOR 32 PROCESSES

Application	Shared Refs. / Total Data Refs. (%)		Private Global Refs. / Total Data Refs. (%)		Private Local Refs. / Total Data Refs. (%)	
	Unopt.	Optim.	Unopt.	Optim.	Unopt.	Optim.
Celm	35.2	61.3	23.1	23.2	41.7	15.4
DWP	28.2	85.1	11.5	2.1	60.3	12.8
Mp3d	43.5	77.8	17.2	0.0	39.3	22.3
LocustRoms	13.9	50.4	15.4	23.5	70.7	26.1
Maxflow	37.0	61.7	6.5	6.6	56.6	31.7
Miscst	11.8	33.7	21.5	37.3	66.6	29.1
AVERAGE	28.3	61.7	15.9	15.4	55.8	22.9

rates for moderate-sized cache blocks. In addition, data layout optimizations are more effective in eliminating false sharing than in improving spatial locality. Overall, these optimizations eliminated about 10% of the misses on shared data.

Our observations on where and how false sharing occurs lead us to hypothesize that false sharing is not the major source of the cache misses in compiler-parallelized code either. For such code, the compiler can easily avoid the obvious false sharing pitfalls. For example, in a *DOALL* loop, it is well known that interleaving individual iterations across different processors can cause false sharing. This effect can be avoided by increasing the granularity of the slices assigned to processors.

Optimizations that improve the performance of cache memories are likely to grow in importance as the latencies of cache misses increase. Of these optimizations, those that specifically optimize the performance of large cache blocks, like the ones presented here, are particularly interesting, since large blocks can be useful in amortizing the cost of a long-latency memory access. More effort should be devoted to optimizing the performance of large cache blocks. In this paper, we have shown data that suggests that researchers should focus on increasing the spatial locality of the data more than on reducing false sharing.

## APPENDIX

Tables VIII and IX classify the data references for 16- and 32-process streams, respectively.

## ACKNOWLEDGMENT

We thank the referees for their helpful comments. We also thank S. Goldschmidt, B. Bray, H. Davis, S. Tjiang and the authors of the applications for their contributions.

## REFERENCES

- [1] D. R. Cheriton, H. A. Goosen, and P. D. Boyle, "Multi-level shared caching techniques for scalability in VMP-MC," in *Proc. 16th Annu. Int. Symp. Comput. Architecture*, June 1989, pp. 16-24.
- [2] J. Elder, A. Gottlieb, C. K. Kruskal, K. P. McAuliffe, L. Rudolph, M. Snir, P. Teller, and J. Wilson, "Issues related to MIMD, shared-memory computers: The NYU ultracomputer approach," in *Proc. 12th Annu. Int. Symp. Comput. Architecture*, June 1985, pp. 126-135.
- [3] J. R. Goodman and P. J. Woest, "The Wisconsin multicube: A new large-scale cache-coherent multiprocessor," in *Proc. 15th Annu. Int. Symp. on Comput. Architecture*, June 1988, pp. 422-431.
- [4] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy, "The directory-based cache coherence protocol for the DASH multiprocessor," in *Proc. 17th Annu. Int. Symp. Comput. Architecture*, May 1990, pp. 148-159.
- [5] G. Pfister, W. Brantley, D. George, S. Harvey, W. Kleinfelder, K. McAuliffe, E. Melton, A. Norton, and J. Weiss, "The IBM research parallel processor prototype (RP3): Introduction and architecture," in *Proc. 1985 Int. Conf. Parallel Processing*, 1985, pp. 764-771.
- [6] A. W. Wilson, "Hierarchical cache/bus architecture for shared memory multiprocessors," in *Proc. 14th Annu. Int. Symp. Comput. Architecture*, June 1987, pp. 244-252.
- [7] A. Agarwal and A. Gupta, "Memory-reference characteristics of multiprocessor applications under MACH," in *ACM SIGMETRICS Conf. Measurement and Modeling of Comput. Syst.*, May 1988, pp. 215-225.
- [8] S. J. Eggers and R. H. Katz, "The effect of sharing on the cache and bus performance of parallel programs," in *Proc. 3rd Int. Conf. Architectural Support for Programming Lang. and Operating Syst.*, Apr. 1989, pp. 257-270.
- [9] W. D. Weber and A. Gupta, "Analysis of cache invalidation patterns in multiprocessors," in *Proc. 3rd Int. Conf. Architectural Support for Programming Lang. and Operating Syst.*, Apr. 1989, pp. 243-256.
- [10] R. L. Lee, P. C. Yew, and D. H. Lawrie, "Multiprocessor cache design considerations," in *Proc. 14th Annu. Int. Symp. Comput. Architecture*, June 1987, pp. 253-262.
- [11] A. J. Smith, "Line (block) size choice for CPU caches," in *IEEE Trans. Comput.*, vol. C-36, pp. 1063-1075, Sept. 1987.
- [12] A. J. Smith, "Cache memories," in *Computing Surveys*, vol. 14, pp. 473-530, Sept. 1982.
- [13] F. Ingoin and R. Triolet, "Supermode partitioning," in *Proc. 15th Annu. ACM Symp. Principles of Programming Lang.*, Jan. 1988, pp. 319-329.
- [14] M. E. Wolf and M. S. Lam, "A data locality optimizing algorithm," in *Proc. ACM SIGPLAN 91 Conf. Programming Lang. Design and Implementation*, June 1991, pp. 30-44.
- [15] F. J. Carrasco, "A parallel maxflow implementation," CS411 Project Rep., Stanford Univ., Mar. 1988.
- [16] J. A. Dykstal and T. C. Mowry, "MINCUT: Graph partitioning using parallel simulated annealing," CS411 Project Rep., Stanford Univ., Mar. 1989.
- [17] A. Galper, "DWF," CS411 Project Rep., Stanford Univ., Mar. 1989.
- [18] J. D. McDonald and D. Baganoff, "Vectorization of a particle simulation method for hypersonic rarified flow," presented at the *AIAA Thermodynamics, Plasmadynamics and Lasers Conf.*, Seattle, WA, June 1988.
- [19] J. Rose, "LocusRoute: A parallel global router for standard cells," in *Proc. 25th ACM/IEEE Design Automation Conf.*, June 1988, pp. 189-195.
- [20] L. Soule and A. Gupta, "Characterization of parallelism and deadlocks in distributed digital logic simulation," in *Proc. 26th ACM/IEEE Design Automat. Conf.*, June 1989, pp. 81-86.
- [21] E. Lusk, R. Stevens, and R. Overbeek, *Portable Programs for Parallel Processors*. New York: Holt, Rinehart, and Winston, Inc., 1987.
- [22] H. Davis, S. Godschmidt, and J. Hennessy, "Multiprocessing simulation and tracing using tango," in *Proc. 1991 Int. Conf. Parallel Processing*, vol. II, Aug. 1991, pp. 99-107.
- [23] F. Baskett, T. Jermoluk, and D. Solomon, "The 4D-MP graphics superworkstation: Computing + graphics = 40 MIPS + 40 MFLOPS and 100,000 lighted polygons per second," in *Proc. 33rd IEEE Comput. Soc. Int. Conf. - COMPCON* vol. 88, Feb. 1988, pp. 468-471.
- [24] M. S. Papamarcos and J. H. Patel, "A low overhead coherence solution for multiprocessors with private cache memories," in *Proc. 11th Annu. Int. Symp. Comput. Architecture*, June 1984, pp. 348-354.
- [25] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. San Mateo, CA: Morgan Kaufmann, 1990.
- [26] G. Kane, *MIPS RISC Architecture*. Englewood Cliffs, NJ: Prentice-Hall, 1989.
- [27] M. D. Hill, "Aspects of cache memory and instruction buffer performance," Tech. Rep. UCB/CSD 87/381, Univ. of California, Berkeley, Nov. 1987.
- [28] S. Przybylski, M. Horowitz, and J. Hennessy, "Performance tradeoffs in cache design," in *Proc. 15th Annu. Int. Symp. Comput. Architecture*, May 1988, pp. 290-298.



**Josep Torrellas** (S'87-M'90-S'91-M'92) received the B.S. degree from the Universitat Politècnica de Catalunya in 1986, the M.S. degree from the University of Wisconsin-Madison in 1987, and the Ph.D. degree from Stanford University in 1992, all in electrical engineering.

He is an Assistant Professor of Computer Science at the University of Illinois at Urbana-Champaign and the Center for Supercomputing Research and Development. His research interests focus on hardware and software techniques to improve the performance of scalable shared-memory multiprocessors.



**Monica S. Lam** (S'84-M'85-S'85-M'86-S'86-M'87) received the B.S. degree in computer science from University of British Columbia in 1980 and the Ph.D. degree in computer science from Carnegie Mellon University in 1987.

She is an Assistant Professor of Computer Science at Stanford University. Her research interests are in developing language, compiler, and architecture technologies that cooperate in exploiting parallelism.

Dr. Lam received an NSF Young Investigator Award in 1992.



**John Hennessy** (S'72-M'77-SM'89-F'91) is a Professor of Electrical Engineering and Computer Science at Stanford University. His current research interests are in exploiting parallelism at all levels to build higher performance computer systems.

Prof. Hennessy was the recipient of a 1984 Presidential Young Investigator Award, and in 1987 was named the Willard and Inez K. Bell Professor of Electrical Engineering and Computer Science. He is a member of the National Academy of Engineering. During a leave from Stanford in

1984-85, he cofounded MIPS Computer Systems where he continues to participate in industrializing the RISC concept as Chief Scientist.

# Analysis of Shared Memory Misses and Reference Patterns<sup>†</sup>

Jeffrey B. Rothman  
Lyris Technologies, Inc.  
2070 Allston Way, Suite 101  
Berkeley, CA 94704  
jeffrey@lyris.com

Alan Jay Smith  
Computer Science Division  
University of California  
Berkeley, CA 94720  
smith@cs.berkeley.edu

## Abstract

Shared bus computer systems permit the relatively simple and efficient implementation of cache consistency algorithms, but the shared bus is a bottleneck which limits performance. False sharing can be an important source of unnecessary traffic for invalidation-based protocols, elimination of which can provide significant performance improvements. For many multiprocessor workloads, however, most misses are true sharing plus cold start misses. Regardless of the cause of cache misses, the largest fraction of bus traffic are words transferred between caches without being accessed, which we refer to as dead sharing.

We establish here new methods for characterizing cache block reference patterns, and we measure how these patterns change with variation in workload and block size. Our results show that 42 percent of 64-byte cache blocks are invalidated before more than one word has been read from the block and that 58 percent of blocks that have been modified only have a single word modified before an invalidation to the block occurs. Approximately 50 percent of blocks written and subsequently read by other caches show no use of the newly written information before the block is again invalidated.

In addition to our general analysis of reference patterns, we also present a detailed analysis of dead sharing for each shared memory multiprocessor program studied. We find that the worst 10 blocks (based on most total misses) from each of our traces contribute almost 50 percent of the false sharing misses and almost 20 percent of the true sharing misses (on average). A relatively simple restructuring of four of our workloads based on analysis of these 10 worst blocks leads to a 21 percent reduction in overall misses and a 15 percent reduction in execution time. Permitting the block size to vary (as could be accomplished with a sector cache) shows that bus traffic can be reduced by 88 percent (for 64-byte blocks) while also decreasing the miss ratio by 35 percent.

<sup>†</sup>Funding for this research has been provided by the State of California under the MICRO program, and by Cisco Corporation, Fujitsu Microelectronics, IBM, Intel Corporation, Maxtor Corporation, Microsoft Corporation, Sun Microsystems, Toshiba Corporation, and Veritas Software Corporation.

## 1 Introduction

Shared memory multiprocessor systems are becoming increasingly popular. The limit to the number of processors that can be placed on the same memory bus is due to the bus traffic demands of the processors. Here we present a new examination of the interference patterns of references to words within shared blocks, with the purpose of aiding both software developers in data layout and hardware designers in the development of new protocols that perform coherence (cache consistency) on a subblock basis. Our purpose is to examine the causes of bad behavior in parallel programs, aiming to reduce bus traffic and miss ratios. This study uses relatively large traces of twelve parallel workloads to provide our results. We measure the sharing behavior of words within shared blocks to determine the extent that false sharing occurs. We also look at the related phenomenon of *dead sharing*, which is determined by measuring the words within a block that are not utilized while in the cache; as will be shown, these words consume the largest proportion of bus traffic.

The remainder of this paper is organized as follows: the next section describes our motivation for undertaking this study. Section 2 provides an overview of related work in the area of characterization of sharing patterns of parallel programs. Section 3 discusses our methodology for creating and evaluating the parallel memory traces and describes some of the metrics we use to measure the underlying behavior that causes shared memory traffic problems. In Section 4 we present our results and discuss our observations. Section 5 summarizes our conclusions.

### 1.1 Definitions

Table 1 provides the definitions of the terms we use throughout this paper.

Definitions Used in This Paper	
<i>Shared Memory</i>	The portion of the memory space that is visible to all processors.
<i>Block</i>	A group of sequential memory locations that are fetched and evicted from caches together, aligned so that the address of first byte of the block has $\log_2(\text{Block Size})$ zeros as the lowest order bits.
<i>Shared Block</i>	A shared block generally is a block from shared memory; more specifically, a block that is referenced by more than one processor during the execution of a program.
<i>Private Block</i>	A block that is accessed only by one processor for the duration of the simulation.
<i>Shared Access</i>	A memory reference to a block that at some point during the simulation is considered to be a shared block. Note: an access that is classified as shared in one simulation may not be shared in a simulation with a smaller block size.
<i>Actively Shared Memory</i>	The set of blocks that are accessed by multiple processors for a given set of simulation parameters.
<i>Global Unshared Access</i>	Access to a portion of memory that is declared to be shared, but is used exclusively by a single processor.
<i>Private Access</i>	A memory reference to a private block.
<i>False Sharing</i>	False sharing occurs when two or more data items that are unrelated happen to be placed in the same block, causing an unnecessary increase in bus traffic to maintain coherence. In this paper we define false sharing as a reference to a word in a block, finding it to be in a different state in a particular block under block coherence and transfer size than under word granularity coherence/transfer size. Thus it can have a good side-effects (such as in the prefetching effect of large blocks), or bad side-effects, such as extra misses and coherency operations.
<i>Dead Sharing</i>	The portions of the block that are not referenced while in the cache, resulting in wasted bus traffic.
<i>Local Read</i>	Read by the processor that has most recently written the block.
<i>Local Write</i>	Write by the processor that has most recently written the block.
<i>Remote Read</i>	Read by any processor except the one that has most recently written the block. This also includes reads by processors with their own copies of the block.
<i>Invalidation Interval</i>	The string of references to a block by all processors between coherence induced invalidations.
<i>Span</i>	Distance between the furthest apart words in a block that are referenced during an invalidation interval.
<i>Processor-spatial</i>	The footprint of accesses by a processor during an invalidation interval.
<i>Cache Hit</i>	The data requested is found in the cache and the processor can proceed without causing a coherence operation.
<i>Fetch Miss (fmiss)</i>	A reference to the cache which does not find the requested data, requiring a fetch from main memory, or another processor's cache.
<i>Invalidation Miss (imiss)</i>	A write reference to a block (or word) held in the cache in the shared state, requiring the processor to stall while invalidating copies of the data in other caches (under sequential consistency).
<i>Miss</i>	Both fetch and invalidation misses.
<i>Reference Run</i>	The stream of uninterrupted references by one processor to a block.
<i>Read Run</i>	A reference run consisting purely of reads
<i>Read/Write Run</i>	A reference run consisting of reads and writes
<i>Write Run</i>	The stream of writes in the read/write run

Table 1. Definitions used in this paper.

## 1.2 Motivation

In the process of determining what sort of memory accesses caused the most traffic in our workloads, we examined the source code to identify which data structures were responsible for the most unfortunate sharing patterns (detailed in Section 4.4 and [20]). Some of the programmer specific details that have come out of this research are described in more detail in Section 4. During our analysis we found several recurring types of data structures which seem to lead to bad data access patterns.

When the programmer is not sufficiently careful in his or her data layout, it is necessary for some combination of the compiler and hardware to try to minimize coherence induced traffic. This paper investigates the sources of traffic caused by inadvertently poor data organization and provides suggestions for solving these problems.

Our goal in this research is to uncover the effects of poor programming style and to provide information about how these problems can be corrected. Additionally, we want to show how all types of sharing impact the performance of these workloads, and to demonstrate the degree to which block size affects spatial locality.

## 2 Background

Previous research on multiprocessor reference patterns has primarily focused on evaluating various cache coherence protocols for suitability, primarily contrasting invalidation- and update-based algorithms. The initial papers in this area ignored block size altogether, exclusively using 4-byte blocks to examine primarily the patterns of writing references [9, 1]. The reference patterns were categorized by the length of the *reference run*. A reference

run can be further refined into *read runs*, *read/write runs*, and the *write run* (Table 1). The write-run lengths varied widely between applications, leading to inconclusive results whether update- or invalidate-based protocols give superior performance.

Research concerning *reference runs* for different block sizes found that for scalar (non-vector) workloads, the lengths of the various reference runs did not increase with block size, although vector workloads showed improved processor locality with larger block sizes [12]. The poor locality in scalar workloads was attributed to fine-grain sharing of data among the processors. A study of the effect of block size on data structures concluded that the excessive invalidations are caused by a mismatch between data objects and block size [15].

To ameliorate the false sharing problems indicated by previous research, several designs have been proposed for multiprocessor systems using a fixed size subblock [14, 2, 5], showing good performance improvement over systems using block size coherence granularity. Variable size block coherence was investigated in [6], showing better performance than fixed size blocks for most workloads, but no implementation details were provided. One dynamically adjustable subblock protocol allowed a block to be divided into two (possibly unequal) pieces for coherence purposes, with a slight improvement in performance [17].

### 3 Methodology

Our work is based on trace-driven simulation (TDS). Initially we used execution-driven simulation for our research, but we changed to TDS for two reasons: (1) the locations of data objects varied as parameters such as block size changed, making detailed analysis very complicated; and (2) our EDS tools are dependent on generally obsolete DEC 5000 machines; using traces allows for much more rapid generation of results on faster PCs and workstations. We used our EDS tool Cerberus [22] to generate traces for the simulation system. The trace generation system simulates synchronization objects (locks and barriers) at run-time, which aids in reducing trace length (by eliminating spin-waiting loops in the trace file) and allows more accurate synchronization behavior while producing traces.

#### 3.1 Simulation

Our simulations use infinite size fully-associative caches to eliminate capacity and conflict misses, to focus on the effect of coherency induced misses and traffic. The remaining misses fall into three categories: cold start/private, cold start/shared, and coherency caused misses. A reference to a private block causes a single cold start miss, which is the only miss for the rest of the simulation; shared blocks can

have up to 16 cold start misses (with 16 processors reading them in for the first time), subsequent shared misses are either caused by true or false sharing. The cache simulators tested parallel workloads with 16 processors and block sizes ranging from 4 to 512 bytes.

#### 3.2 Workload Characterization

Twelve parallel programs were examined to provide the results for this paper. Ten of the programs come from the SPLASH suites from Stanford University, which have been available to the research community as a *de facto* benchmark for comparing parallel program execution. We use some of the older SPLASH workloads because we believe these to be more typical programming efforts, rather than the somewhat more optimized SPLASH2 workloads. These programs have all been used in a number of studies analyzing parallel code performance and are characterized and described in more detail in [23, 26]. The other two programs used in this study (*topopt* and *pverify*) were created by the CAD group at U.C. Berkeley and have been used for measurements at Berkeley and the University of Washington [10, 11, 8, 3]. Detailed descriptions of the workloads can be found in [20]. All workloads were traced on a MIPS R3000 based workstation using the Cerberus multiprocessor simulator [22]. Each workload is traced from beginning to end to capture the entire behavior of the program.

Table 2 shows the reference characteristics for a 16 processor 4-byte block simulation of the various workloads on our SMP simulator, which runs on uniprocessor workstations. The number of shared references in Table 2 were measured using 4-byte blocks, which captures the number of truly shared words. Global unshared references and private references (Table 1) are lumped together under the *private* heading. The fraction of shared accesses has quite a large variation; it ranges from 0.16 in *barnes* to 0.90 in *topopt*, with an average of 0.43 for all workloads. However, as the block size increases, memory locations and references which are classified private in Table 2 can become shared, so it is necessary to trace all references which are to shared memory. We also tracked all references to private memory to understand its contribution to total memory traffic. As the cache simulators were designed to make the common transactions very quick through the use of hashing, tracking the references to private memory is generally not a major contributor to simulator execution time.

#### 3.3 Metrics

The traditional reference stream interval used for measuring sharing behavior is the *reference run* [9, 1, 12]. Along with related measures such as the *read run*, *read/write run*, and the *write run* (Table 1), it can provide

Program Characteristics									
Programs	References (Millions)		Shared Data	Private Data	Shared			Private	
			(KBytes)		Fraction of Data References				
	Inst	Data			Reads	Writes	Locks	Reads	Writes
barnes	114.23	42.31	33.02	34.64	0.159	0.005	0.002	0.456	0.379
cholesky	90.67	34.32	970.02	783.38	0.496	0.075	0.013	0.287	0.130
fmm	288.30	166.82	380.41	460.14	0.136	0.006	0.000	0.347	0.511
locus	805.62	164.45	1405.70	1151.79	0.563	0.020	0.001	0.255	0.162
mp3d	174.88	60.82	701.91	181.53	0.318	0.223	0.001	0.302	0.157
ocean	234.12	92.38	140.16	984.45	0.264	0.031	0.014	0.560	0.132
pthor	275.87	97.77	1233.09	1026.75	0.384	0.047	0.049	0.350	0.176
pverify	181.29	55.24	23.08	149.67	0.473	0.015	0.008	0.320	0.186
raytrace	471.13	196.96	667.30	2144.09	0.318	0.003	0.002	0.429	0.248
topopt	655.75	141.60	19.22	38.76	0.812	0.087	0.000	0.085	0.016
volrend	351.62	79.92	395.61	2340.98	0.477	0.007	0.003	0.287	0.226
water	366.23	127.67	44.51	102.37	0.179	0.016	0.002	0.577	0.227
Total					Average				
Overall	4009.7	1260.3	6014	9399	0.381	0.044	0.008	0.354	0.213

Table 2. Reference characteristics of workloads for 16-processor simulation, using 4-byte blocks.

some idea of the residency time of a block in one processor's cache (processor locality) and the appropriateness of coherence protocol (invalidate vs. update).

A major problem with the reference run as a metric is the lack of information concerning how the processors share data within blocks. It is possible to get a crude idea of contention by examining the length of these different types of reference runs, but they provide no indication about the type or granularity of sharing within a block. We establish a new reference stream interval called an *invalidation interval*, which is the string of references to a block by all processors between coherence induced invalidations (Figure 1). This allows a longer term and much more detailed study of dynamic sharing behavior within a block.

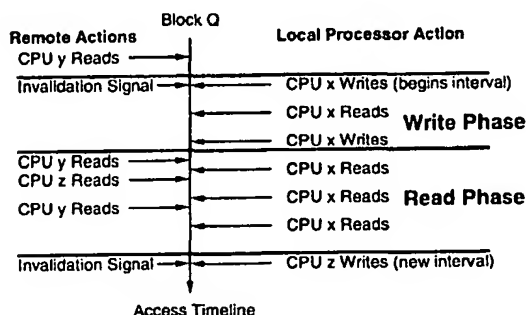


Figure 1. An invalidation interval is a string of references to a block, lasting from the first invalidation to a block until the next invalidation. During the interval other processors may read the block, but not write it.

The metrics we describe here are all concerned with the *processor-spatial* properties of multiprocessor programs. By this term we mean the number of unique words within a block that are accessed while it is valid in a cache. This concept also encompasses measuring the fraction of the block which contains these words, which we refer to as the *span*. By using these types of metrics, it is possible to get an idea of the typical range of block sizes that make sense to use with multiprocessor caches.

The results presented later in this paper demonstrate that a variable block size (or use of subblocks) can significantly outperform any fixed block size for all the workloads examined, reducing traffic by 88 percent while decreasing the miss ratio by 35 percent (on average) for 64-byte blocks.

We can think of an invalidation interval as having two phases: (1) a write phase and (2) a read phase. The write phase consists of local reads and writes, which cause no bus activity after the first write (assuming write-allocate). The read phase (if there is one) begins with the first remote read, and consists of local and remote reads. We use the statistics of the references to individual words during invalidation interval to evaluate processor-spatial locality, which we will show is generally rather poor, i.e., large block transfers for shared data are demonstrated to be wasteful.

Our goal is to provide the tools to aid in improving spatial locality in shared memory systems, or at least provide insight into the lack of spatial locality. It has been demonstrated that shared data in multiprocessor workloads have worse locality of reference than unshared data [11], but increasing the block and cache sizes have not always provided a solution. It is necessary to understand what kind of misses are causing poor performance and examine the data structures/objects that produce such problems.

Implicitly our study assumes a write-invalidate protocol as backdrop against which our analysis is done. Invalidation-based protocols logically offer a better solution to bus-based systems, due to the necessity of reducing traffic over the shared bus to memory. A pure update protocol (update on each write to a shared block) uses an estimated 2-25 times the traffic of write-invalidate protocols for coherence related operations [19], and the amount of network traffic increases with cache size. This is caused by the requirement to update on each write to shared cache blocks, regardless of the age (staleness) of the block in the cache; this problem worsens as the number of blocks in the cache increases, generating the most bus traffic for infinite caches. There are some adaptive protocols that allow switching between update and invalidate for each block depending on the access pattern for the block; but of the non-adaptive coherence protocols, invalidation-based protocols typically outperform update-based protocols [13]. Additionally, write-invalidate protocols are the most popular class of protocols that are actually implemented in real systems [24, 16], which makes them a more attractive target for performance improvement. When a program is properly (re)structured to reduce the movement of blocks between processors, write-invalidate based protocols provide better overall performance.

## 4 Results

This study examines SMP (symmetric multiprocessor) memory access behavior on three levels, which successively refine the granularity of the inspection to smaller features. The first and coarsest level of analysis looks at the aggregate behavior of all of the memory references. By an analysis of total bus traffic, we show that **dead sharing** traffic is the dominant factor for block sizes greater than 16 bytes. We also examine false-sharing, extending our analysis to two kinds of misses: those that require data transfers (fetch misses or fmisses), and those that require only coherence transactions (invalidation misses, or imisses). Our analysis of false sharing misses is found in [20]; it has been omitted here for brevity, and because much of the analysis of false sharing has already appeared elsewhere in the literature [8, 7, 25].

The second level of memory reference observation looks at the spatial reference pattern to shared blocks. This consists of examining the unique (distinct) words within a block that are referenced from the time it is read into the cache until the time it is invalidated. In addition, we look at the footprint of those words within the block, which we refer to as the **span**. The span provides a means of determining the spatial locality of a set of block references.

To develop a hardware protocol or software restructuring method to reduce bus traffic from coherence overhead, it is

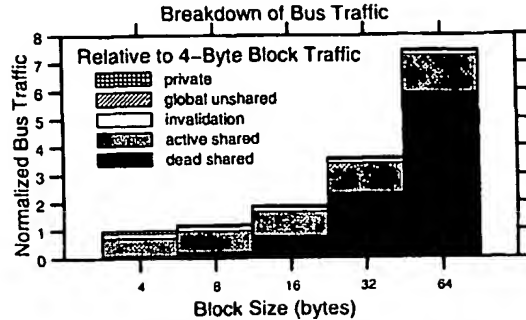


Figure 2. Breakdown of workload average bus traffic, normalized to 4-byte block traffic.

necessary to understand the patterns of sharing that occur and the competition of processors for words within shared blocks. At our finest grain level of examination, the words from the 10 worst (judged by misses) 64-byte blocks from each workload are characterized by the reference pattern for each individual word. This provides insight into the types of data objects which cause much of the traffic problems when they are placed in close proximity and provides hints into hardware and software solutions that can be used to eliminate or ameliorate much of the traffic/miss problem.

### 4.1 Dead Sharing

In an attempt to measure the impact of large block sizes on bus utilization in an implementation independent manner, we use the bus traffic metric. Each transaction transmits a 4-byte address across the bus plus (when appropriate) some number of data bytes. The bus traffic consists of fetches (address + fmisses  $\times$  block size) and invalidations (consisting only of the fixed overhead to transfer an address over the bus, since no data transferred is required). This is a reasonable estimate for bus utilization for split-transaction busses. Figure 2 shows average bus traffic for infinite caches with 4- to 64-byte blocks (relative to 4-byte block traffic), broken down into 5 main types of traffic: private traffic, global unshared traffic, invalidation signals (address transfer only), active shared traffic (truly utilized) and dead shared traffic.

The dead shared traffic is determined by analyzing which words in a shared block have not been accessed at the time the block is invalidated. The active shared portion of the shared traffic consists of the words that were actually referenced before the block was invalidated. The breakdown shows that traffic from private blocks is relatively insignificant (from 1.17 percent for 4-byte blocks to 0.10 percent for 64-byte blocks on average) and traffic from global unshared blocks starts at 9.3 percent and declines to 0.5 per-

cent for 64-byte blocks. Dead sharing traffic causes about 41.0 percent of the traffic with 16-byte blocks and grows very rapidly with larger block sizes. The traffic approximately doubles for each increase in block size beyond 64-byte blocks, reaching 54.3 times 4-byte block traffic when 512-byte blocks are used. The active shared traffic increases much more slowly than dead shared traffic. Increases in the active shared traffic are due to the incorporation of global unshared data into shared blocks as the block size increases, so that global unshared references are turned into active shared references, causing more active shared traffic. Dead sharing traffic hits 79.3 percent of total traffic with 64-byte blocks and reaches 95.1 percent with 512-byte blocks. This indicates that there is much room for enhancing the operation of shared memory systems.

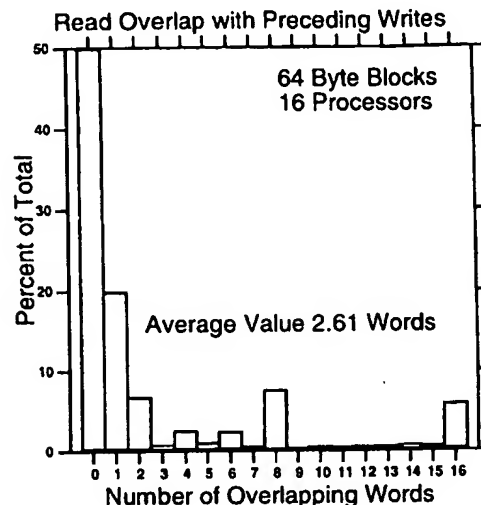
Dead sharing traffic results from both false and true sharing that causes a block to be invalidated before all the words within the block can be utilized. Active/dead sharing and true/false sharing are somewhat orthogonal concepts; active/dead sharing deals with processor-spatial locality, true/false sharing concerns the sharing of words between processors. However, blocks exhibiting false sharing behavior are also going to generate significant dead sharing traffic. The next section looks at the access patterns of distinct words within the blocks to understand the cause of this dead sharing.

Note that when trying to establish statistics like bus performance for a realistic system, there would be a number of parameters to consider. For example, bus utilization is a better metric than bus traffic for measuring how close to saturation the bus is. In such a case, implementation dependent issues must be considered, such as bus width, actual transaction overheads, bus pipelining, memory latency, split vs. non-split transaction, etc. Bus utilization and saturation issues are beyond the scope of this paper but are considered in [21]. Assuming a one cycle address transfer time and a split-transaction bus, the bus traffic information in Figure 2 shows a reasonably good approximation of relative bus utilization between various block sizes. In a memory system that does not support split-transactions, the bus would be unusable during the memory latency period as well, which would cause a dramatic change in the relative bus utilization from what is displayed here.

## 4.2 Granularity of Sharing

When the set of words being written to a block by one processor shows little correspondence to the words read by other processors, there is a strong indication that false and dead sharing are a problem. For example, if generally one word in a block is the target for all writes to the block, but many of the other words are read-only after initialization (as occurs in the *GlobalMemory* (or similarly named) data

structure used for global shared variables in many of our workloads), most of the data in that block is needlessly invalidated almost every time a write occurs.



**Figure 3. Read-write sharing during invalidation intervals, average of all workloads.**

Figure 3 shows a histogram of the number of reads that overlap preceding written words from the same invalidation interval for 64-byte (16-word) blocks, averaged over all workloads. Roughly 50 percent of invalidated cache blocks have no overlap of the words read in the second phase of an invalidation interval with the series of writes that began the invalidation interval, meaning that none of the updated information was accessed before another cache miss occurs for the processors reading the block. Approximately 20 percent of blocks have a read-write overlap of only a single word. The number of updated words read before an invalidation rapidly falls off, but with significant components for 8 and 16 words. These statistics demonstrate that half of the invalidations are caused by updates to words which are not subsequently read by other processors before the blocks are again invalidated, fully wasting the information transfer. A large fraction of those blocks which do read updated words only read a single word before invalidation. Increasing the block size affects the degree of overlap by increasing the likelihood that writes by different processors prematurely invalidate information in the block, which causes the average overlap to peak at 2.7 words for 128-byte blocks and drop with larger block sizes [20], indicating a massive waste of bus traffic.

Another method to examine how words within blocks are shared between processors is to measure whether writes by different processors to the same block happen to occur

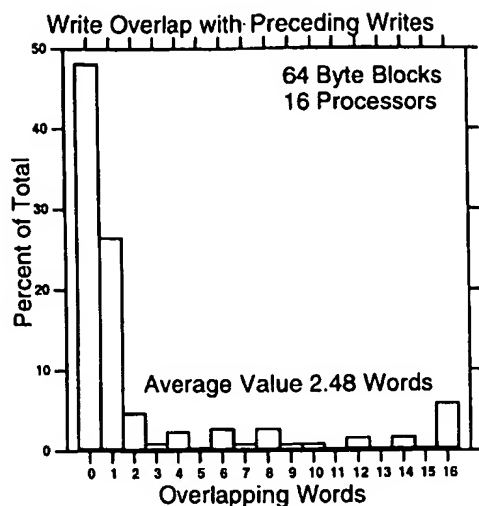


Figure 4. Write-write sharing between invalidation intervals, average of all workloads.

to the same set of words (write-write sharing granularity). Figure 4 shows a breakdown of the average case for 64-byte blocks, calculated by logically ANDing the vector of words written during successive invalidation intervals when the writers have different processor IDs. Almost 50 percent of the succeeding intervals which have different processors writing show no overlap in the set of words written. Of the remaining blocks which have at least one written word in common, the next largest value shows an overlap of only a single word. The histogram indicates a bifurcation of access patterns to blocks: either succeeding processors accessing a block have very little write sharing, or share a large fraction of the block (the 16 word component is somewhat prominent). Figures 3 and 4 indicate that a coherency protocol that could adaptively choose large or small granularity for enforcing coherence could be very successful in reducing coherency traffic.

### 4.3 Spatial Locality

A typical way to measure spatial locality for uniprocessors is to examine the change in the miss ratio as the block size varies. A more insightful metric for multiprocessors is our characterization of the read and write references to blocks between invalidations. A type of spatial locality (which we refer to as *processor-spatial locality*) can be determined by measuring the span and number of distinct words (defined in Table 1) referenced in a block while the block is valid in the cache, regardless of the interleaving read accesses by other processors. For each invalidated block, the words within the block are examined to see how

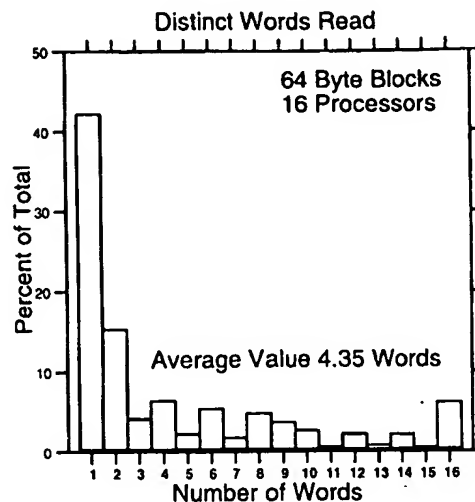
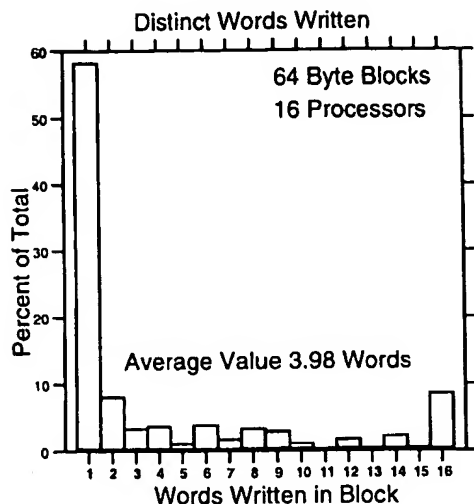


Figure 5. Distinct words read before invalidation, average of all workloads.

well they were utilized. If too many of the fetched words are not used, bandwidth is wasted in the form of dead sharing. Likewise, if too many words in the invalidated block are not subsequently updated, valid data was needlessly canceled and likely will have to be re-read, also a waste of bus bandwidth. An increase in block size increases the likelihood that a remote write to unrelated data can unnecessarily invalidate a block from the cache, causing dead sharing.

To provide a more detailed view of spatial locality, Figures 5 and 6 show the number of distinct words read within a block and the number of words written to a block while the block is in the cache, respectively, for 64-byte blocks, averaged over all workloads. This shows that the plurality (42 percent) of the blocks are invalidated by another processor before more than one distinct word is read, and only a single word is written 58 percent of the time during an invalidation interval. Except for single word blocks that have 100 percent usage (due to demand fetching), the percentage of blocks with only a single word read or written before a block is invalidated holds in a narrow range over block size, ranging on average between 37.6 and 45.8 percent for reads, and between 48.5 and 64.0 percent for writes [20]. The data shows that not much of the information in a block is used in any manner before another processor interferes, either by causing an invalidation of the data (in the case of a remote write), or by reading the block (placing it in the *shared* state), forcing the next write to cause an invalidation miss.

These figures indicate that it is not often that a whole block need be invalidated, since frequently only a single



**Figure 6. Distinct words written during an invalidation interval, average of all workloads.**

word is written before another invalidation occurs, which could be caused by a remote write, or a remote read followed by a local write. For many of the workloads, the histograms of the number of distinct words read/written (Figures 5 and 6) are very similar in composition to the span of the words read/written [20], indicating a great deal of spatial locality to the portions of the block that are used. To understand the underlying cause of the poor block usage, we next examine the memory reference patterns that cause dead and false sharing to occur.

#### 4.4 Examination of Problem Blocks

Many of the dead and false sharing problems can be directly linked to poor programming style or ignoring the role of the cache in shared memory systems. Although caches are designed to be invisible to the programmer, poor data placement can have a large effect in reducing system performance.

Table 3 shows statistics about the 10 worst behaving blocks (based on the number of fetch misses) for each of our 12 workloads. For each workload, we specify the fraction of actively shared memory these blocks occupy, the fraction of total shared references to these blocks, the fraction of false and true sharing fetch and invalidation misses for which these blocks are responsible, and a classification of the reference patterns of the words within the blocks. The categorization of the words within the blocks are partially based on reference patterns classified in [15, 4]. The 10 worst blocks are (on average) responsible for a good deal of the false sharing misses as well as many of the true

sharing misses. The number of misses are far out of proportion to the number of memory references and the fraction of shared memory space these blocks occupy. The basic problem is that variables are placed (perhaps inadvertently during dynamic memory allocation) into the same blocks as variables or data structures with incompatible reference patterns (for example, arrays of private read-write variables that are accessed by processor ID, read-only variables next to frequently written variables, etc). Almost one-fourth of the words in these blocks are locks with low contention (i.e., not much competition for them), that in isolation would cause little problem, but interact poorly together because locks have poor processor-spatial locality of reference. Other problem words are broadcast words (one processor writing, many processors reading) that cause false sharing misses when placed together (but are still likely to have a fair number of true sharing misses in isolation), and read-write variables that interact together poorly.

In [20] we provide a more detailed analysis of each of the workloads to determine the kind of data structures that are causing most of the problems. Here we present a summary of our attempts to restructure 4 of the workloads ourselves, and some programming hints to prevent such problems in the future.

##### 4.4.1 Improvements

Four workloads (**barnes**, **pthor**, **topopt**, **water**) were chosen to be restructured in an attempt to improve program performance. We modified the workloads to repair problems observed in the worst ten (64-byte) blocks of each. The details of the changes made to the workloads, and the data structures associated with the 10 problem blocks for each of the workloads can be found in [20].

When modifying the data structures involved with the worst 10 blocks, some of these changes carry over to other blocks not in the top 10. However, improvements to the worst 10 blocks could also worsen behavior in other blocks. For example, isolating pieces of data structures by placing them in their own blocks (by adding unused arrays of integers to pad-out the members), can cause misses to increase for many well performing blocks with the same data layout, causing lackluster improvement in the number of misses.

For most of the restructured programs, the number of instructions increases slightly. This is generally due to the extra pointer dereferencing required by isolating per-processor data in separate data structures. On average, the number of instructions increases by less than 1.5 percent, while reducing the number of misses by more than 20 percent (Table 4). Although it depends on how much of a problem the data miss ratio is to begin with, this appears to be a reasonable trade-off. Using a multiprocessor timing simulator with 24 processor cycle memory latency sup-

Categorization of Words Within the 10 Worst Blocks																	
Program	10 Worst Blocks						Reference Types (percent of words)										
	% Shared		% Fmisses		% Imisses		Private				Shared						
	mem	refs	False	True	False	True	un	pl	pro	prw	hl	ll	ro	rm	mi	br	rw
barnes	1.62	4.7	22.7	18.7	11.1	26.7	14.4	0.0	0.0	0.0	0.6	31.9	0.0	1.9	1.2	48.1	1.9
cholesky	0.06	11.4	75.2	6.1	37.3	0.5	6.2	0.0	0.0	3.1	0.0	88.1	0.6	0.6	0.0	1.2	0.0
fmm	0.12	0.1	8.4	4.8	0.2	2.4	0.6	0.6	0.0	0.0	1.2	17.5	0.0	0.0	0.0	0.0	80.0
locus	0.03	8.2	53.0	1.9	20.3	5.9	30.0	6.2	1.2	20.6	1.2	8.8	2.5	0.0	10.0	14.4	5.0
mp3d	0.08	2.6	9.1	5.3	2.9	4.7	11.9	0.0	1.2	2.5	0.6	1.2	3.8	0.6	0.0	10.6	67.5
ocean	0.14	4.7	20.9	35.2	3.8	25.9	10.0	0.0	0.0	32.5	2.5	0.0	36.2	3.8	0.0	15.0	0.0
pthor	0.03	30.5	63.4	49.4	36.1	38.4	16.9	0.0	15.6	1.2	1.2	12.5	6.2	0.0	0.0	31.2	15.0
pverify	0.52	7.4	29.5	5.7	29.9	0.2	10.0	0.0	1.2	61.2	0.0	10.0	6.2	0.0	0.0	10.6	0.6
raytrace	0.05	1.0	56.4	52.2	67.6	83.8	46.2	0.0	8.8	0.6	1.2	10.6	7.5	0.0	0.6	15.0	9.4
topopt	1.27	18.0	65.2	8.0	92.3	7.6	10.0	0.0	0.0	2.5	0.0	0.0	0.0	0.0	45.6	33.1	8.8
volrend	0.06	9.8	50.5	5.6	0.1	18.3	70.0	0.0	0.0	0.0	1.2	10.6	1.2	3.1	0.0	8.1	5.6
water	1.04	0.9	100.0	26.6	0.0	0.7	7.5	0.0	0.0	0.0	1.2	91.2	0.0	0.0	0.0	0.0	0.0
Average	0.4	8.3	46.2	18.3	25.1	17.9	19.5	0.6	2.3	10.3	0.9	23.5	5.4	0.8	4.8	15.6	16.2

**Table 3. Categorization of the word reference patterns of the worst behaving 64-byte blocks for each workload. The categories for classifying each word in the worst blocks consist of: un- unused, pl - private locks, pro - private read-only, prw - private read-write, hl - high contention locks (> 3 seeking access), ll - low contention locks (3 or fewer processors seeking the lock on average), ro - shared read-only, rm - shared read-mostly (at least 75 percent of references are reads), mi - migratory (more than 6 uninterrupted references on average by each processor accessing it), br - broadcast (one processor writing, many processors reading), and rw - read-write (words that do not fall into the other categories).**

porting split transactions, 4-byte memory path (4 processor cycles per word), 4-byte memory addresses for each memory transaction, and 4 processor-cycle bus arbitration, we present (Table 4) the effects of the optimizations for two cache sizes (16K and 64K bytes per processor for 16 processors). Since the effects on the miss ratios reported in Table 4 are for infinite caches with single cycle memory accesses, effects such as capacity misses are not included. (The timing simulations, in the last two columns, are based on finite cache sizes.) When using small caches, the capacity misses can overwhelm the coherence misses, possibly worsening behavior if spatial locality is disturbed too much. The end result of the optimizations reduced execution time by approximately 15 percent. In the case of **topopt**, which spends most of its time waiting for memory, the spatial locality was increased at the same time that false sharing was reduced, leading to a tremendous increase in performance. In the next section, we present some programming hints; however, these optimizations must also be reconciled with the impact they have on spatial locality and capacity misses.

#### 4.4.2 Programming Hints

Based on the detailed examination of the problem areas of our workloads, we provide here a distillation of the poor programming choices that lead to so many false sharing misses: high contention locks should be isolated from each

other and from all other data; in many programs they are kept in arrays. Low contention locks should be placed with the data they protect ([18] also found co-location of locks and data in the same block to be generally beneficial). Some arrays (regardless of data type) are accessed using the ID of the processor as the index into the array; in some cases this results in a group of essentially private read-write variables being assigned to the same block, causing a large quantity of false sharing misses and dead sharing traffic.

Sometimes variables that appear to have true sharing misses can be restructured to eliminate almost all misses. For example, in **pthor** each processor accesses a particular shared variable only to increment the value. The value is actually only used in extremely rare cases (none that we observed during program execution), but the incrementation by each processor causes many true sharing misses. This variable can be restructured to isolate private copies for each processor, to be summed up when the value is actually needed. By examining program behavior more carefully using tracing and by programming with cache coherence in mind, significantly higher performance can be obtained.

#### 4.5 Proper Block Sizing

To demonstrate the performance improvement that can be obtained by reducing false and dead sharing, we use data collected from trace driven simulations of each program to

Data Restructuring to Reduce Coherence Misses (64-byte Blocks)					
Program	Infinite Cache, Single Cycle Memory			Realistic System	
	Overall Miss Reduction	False Sharing Miss Reduction	Instruction Increase	Execution Time	
				16K cache	64K cache
barnes	12.0%	21.1%	0.0%	-1.4%	-6.7%
pthor	21.4%	53.3%	0.3%	-7.2%	-9.5%
topopt	31.9%	30.1%	3.9%	-42.3%	-46.0%
water	20.0%	99.9%	0.2%	-9.0%	-6.0%
Average	21.3%	51.1%	1.1%	-15.0%	-15.4%

Table 4. Results of rearranging data structures to reduce coherence misses, 64-byte blocks.

find the best block size for each individual word in the memory space. Each program was simulated with block sizes from 4 to 512 bytes. For each shared word, we kept track of the address tag, the number of imisses and fmisses, and various other statistics. Using a simple greedy algorithm designed to minimize bus traffic, we demonstrate that a cache that supports multiple block sizes significantly outperforms all fixed-block systems.

The heuristic algorithm that is used to select the block sizes is designed to minimize bus traffic through the use of variable (static) size blocks; i.e., the block size choice varies over the memory space of the program, but any given word is assigned to a specific fixed block size for the entire program execution. Included as part of the bus traffic is a 32-bit (4-byte) address for each bus transaction (both imisses and fmisses) and the data transferred over the bus (fmisses only). Figure 7 shows the process by which blocks are evaluated for the best size. Starting with each word in the memory space that is used, neighboring blocks are combined if when combined they produce less bus traffic than when left as single blocks. When neighboring words have similar access patterns and it is useful to prefetch one while demand fetching the other, the traffic is reduced when the words (or blocks) are grouped into a single unit due to fewer address transmissions over the bus. When excessive traffic is generated due to false or dead sharing, the problem blocks are isolated by not combining them into larger units. The combining process continues until the maximum block size of 64 bytes is reached.

Figure 8 shows the fixed (uniform) block size simulation performance, normalized to the performance of our heuristic (the line across the lower two graphs at value 1.0; note the log scale on the vertical axis). The heuristic uses less traffic than any fixed block size for all workloads, sometimes as much as 47 times better in the most extreme case (pverify). On average it has 87.8 percent less traffic than a 64-byte fixed size block. At the same time, the number of misses is reduced by an average of 35.2 percent. In the worst case it is still within a factor of two of misses for fixed 64-byte blocks (volrend). Compared with 4-byte fixed size

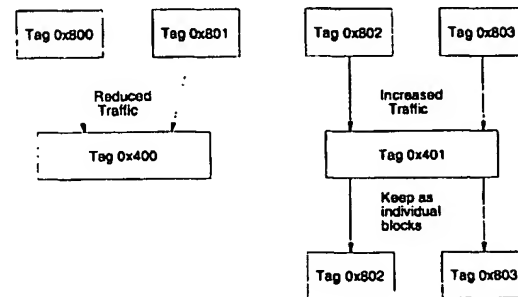
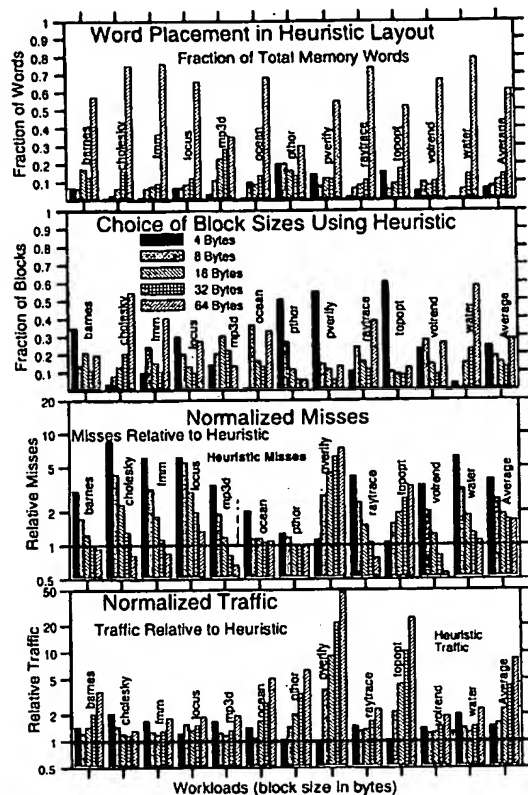


Figure 7. Heuristic algorithm: neighboring blocks are combined into a larger block when it reduces traffic, otherwise the blocks are left at their current size.

blocks, the heuristic has 70.4 percent fewer misses and 23.8 percent less traffic. Note that other heuristics are possible; for example, one could try to minimize the miss ratio rather than the bus traffic. Timing simulations would be required to determine which heuristic performs the best, but we believe that reducing traffic (while not increasing misses) on a shared bus system is a reasonable simple target, given that bus utilization is typically the bottleneck, and that bus traffic correlates with cache misses, and therefore CPU idle.

The block sizes chosen using our heuristic (the diagram second from the top of Figure 8) are most frequently 4-bytes and 64-bytes, with 8-byte blocks slightly less popular. That these two extremes are most popular is not surprising, based on the results from previous sections. Large blocks are best for shared regions with high processor locality; small blocks work best for regions in which there is a high probability that adjacent words are in use by different processors. Note, however, that in general there is a large variation between the optimal block sizes between the different workloads. We can also see that when the number of blocks of each size is multiplied by the size of the blocks, most words are still included in the bigger blocks (top of Figure 8). From the results shown here, we conclude that: (1) the use of



**Figure 8. Normalized traffic and misses for fixed sized blocks normalized with respect to the variable block size heuristic and the choice of block sizes the heuristic uses. Note the log scale on the y-axis on the lower two graphs.**

variable block sizes permits the system to compensate for a mixture of false sharing and high processor-spatial locality; (2) alternately, it should be possible for the programmer to rewrite his or her code to avoid many false sharing situations (Table 3). Note that the method we have used for this analysis would generally be of very little use in a real computer system, since applying it would require that programs be traced and analyzed, and that each block of the program address space be tagged (or otherwise identified) with a block size. It might be possible to have the compiler do some static analysis, and associate block sizes with regions, but the effectiveness of that approach has not been considered here. The purpose of our analysis, rather, has been to identify a promising direction for improvement.

In a follow-on paper [21], we use the results of this research to develop an invalidation-based cache coherence

protocol that uses dynamically-sized subblocks for fetching and invalidation. By tracking the pattern of writes to a block between remote events to the block, the smallest subblock with a power-of-two number of words that contains the modified words is used as the subblock size. The subblock size is reevaluated occasionally, and adjusted to the most commonly measured value. Using variable subblock sizes, we find that our protocol outperforms a regular full block coherence protocol for all workloads, reducing the execution time by 35 percent (on average), as well as outperforming fixed size subblock protocols.

## 5 Conclusions

In this paper we have analyzed shared memory misses and bus traffic at three levels: in aggregate, statistically as words within blocks during the invalidation interval, and by examining special/bad cases in fine detail. The bulk analysis of misses shows that false sharing is generally not the largest fraction of the total misses for most workloads, being fewer than cold start and true shared misses. When analyzing the traffic caused by cache coherence, we do find that a significant problem is the fraction of bus traffic that is transferred between caches without being accessed, which we refer to as **dead sharing**.

Our analysis of invalidated blocks shows that typically only a small fraction of a block is referenced before it is invalidated. Generally there is little or no overlap between the regions of cache blocks updated by writing processors and read by other processors between invalidations to the blocks. Processors writing to the same block show very little overlap in most situations, but a great deal of overlap in a significant number of occasions. From this analysis we believe a good case can be made for adaptively detecting the granularity of sharing within individual blocks and appropriately adjusting the portion of the block that is invalidated.

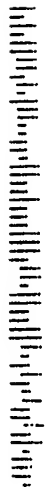
False sharing (and to some degree true sharing) shows a tremendous degree of concentration. The ten blocks with the highest number of misses from each workload contain close to half of all false sharing misses on average and a large number of true sharing misses and invalidations. These blocks generally take up a tiny fraction of the shared memory space and a small fraction of total data references. By looking at the reference patterns of each of the individual words within the offending blocks we found a large problem with arrays of locks and arrays of otherwise private words that exhibit classical false sharing. Another significant problem was frequently accessed read-only variables placed in proximity to write-shared variables. The concentrated nature of bad behavior indicates that a little attention to detail by the programmer would go a long way towards reducing misses and significantly improving performance;

our efforts led to a 21 percent decrease in total misses, resulting in a 15 percent decrease in simulated execution time.

We examined a simple greedy algorithm heuristic which determined the best size block with which each word in main memory should be associated. Based on the results of this heuristic, we find that by using a variety of block sizes, bus traffic can be reduced a significant amount over 64-byte fixed size blocks while generally reducing miss ratios. Many of the best choices of block sizes for improving performance using our heuristic were 4- and 8-byte blocks (due to false and dead sharing), yet most of the data should be placed in larger blocks. This indicates that a cache that supports variable granularity fetching and invalidation (i.e., judicious use of subblocks) should greatly enhance program performance.

## References

- [1] Anant Agarwal and Anoop Gupta. Memory-Reference Characteristics of Multiprocessor Applications under MACH. In *Proc. 1988 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 215–225, Santa Fe, NM, May 24–27 1988.
- [2] Craig Anderson and Jean-Loup Baer. Design and Evaluation of a Subblock Cache Coherence Protocol for Bus-Based Multiprocessors. Technical Report TR-94-05-02, University of Washington, May 1994.
- [3] Craig Anderson and Jean-Loup Baer. Two Techniques for Improving Performance on Bus-based Multiprocessors. In *Proc. First IEEE Symposium on High-Performance Computer Architecture*, pages 264–275, Raleigh, NC, January 22–25 1995.
- [4] John B. Carter, John K. Bennett, and Willy Zwanepeel. Techniques for Reducing Consistency-Related Communication in Distributed Shared-Memory Systems. *ACM Transactions on Computer Systems*, 13(3):205–243, August 1995.
- [5] Yung-Syau Chen and Michel Dubois. Cache Protocols with Partial Block Invalidations. In *Proc. Seventh International Parallel Processing Symposium*, pages 16–23, Newport, CA, April 13–16 1993.
- [6] Czarek Dubnicki and Thomas J. LeBlanc. Adjustable Block Size Coherent Caches. In *Proc. 19th Annual International Symposium on Computer Architecture*, pages 170–180, Gold Coast, Queensland, Australia, May 19–21 1992.
- [7] Michel Dubois, Jonas Skeppstedt, Livio Ricciulli, Krishnan Ramamurthy, and Per Stenström. The Detection and Elimination of Useless Misses in Multiprocessors. In *Proc. 20th Annual International Symposium on Computer Architecture*, pages 88–97, San Diego, CA, May 16–19 1993.
- [8] Susan J. Eggers and Tor E. Jeremiassen. Eliminating False Sharing. In *Proc. 1991 International Conference on Parallel Processing*, pages 1–377–1–381, St. Charles, IL, August 12–17 1991.
- [9] Susan J. Eggers and Randy H. Katz. A Characterization of Sharing in Parallel Programs And Its Applicability to Coherency Protocol Evaluation. In *Proc. 15th Annual International Symposium on Computer Architecture*, pages 373–382, Honolulu, HI, May 30–June 2 1988.
- [10] Susan J. Eggers and Randy H. Katz. Evaluating the Performance of Four Snooping Cache Coherency Protocols. In *Proc. 16th Annual International Symposium on Computer Architecture*, pages 2–15, Jerusalem, Israel, May 28–June 1 1989.
- [11] Susan J. Eggers and Randy H. Katz. The Effect of Sharing on the Cache and Bus Performance of Parallel Programs. In *Proc. Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 257–270, Boston, MA, April 3–6 1989. ACM.
- [12] Jeffrey D. Gee and Alan Jay Smith. Analysis of Multiprocessor Memory Reference Behavior. In *Proc. 1994 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 53–59, Cambridge, MA, October 10–12 1994.
- [13] Jeffrey D. Gee and Alan Jay Smith. Evaluation of Cache Consistency Algorithm Performance. In *Proc. Fourth International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 236–248, San Jose, CA, February 1–3 1996.
- [14] James R. Goodman. Coherency For Multiprocessor Virtual Address Caches. In *Proc. 2nd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 72–81, Palo Alto, CA, October 5–8 1987.
- [15] Anoop Gupta and Wolf-Dietrich Weber. Cache Invalidation Patterns in Shared-Memory Multiprocessors. *IEEE Transactions on Computers*, 41(7):794–810, July 1992.
- [16] John Hennessy and David A. Patterson. *Computer Architecture, A Quantitative Approach*. Morgan-Kaufmann, 2nd edition, 1996.
- [17] Murali Kadiyala and Laxmi N. Bhuyan. A Dynamic Cache Subblock Design to Reduce False Sharing. In *Proc. International Conference on Computer Design: VLSI in Computers and Processors*, pages 313–318, Austin, TX, October 2–4 1995.
- [18] Alain Kägi, Doug Burger, and James R. Goodman. Efficient Synchronization: Let Them Eat QOLB. In *Proc. 24th Annual International Symposium on Computer Architecture*, pages 170–180, Denver, CO, June 2–4 1997.
- [19] David J. Lilja. Cache Coherence in Large-Scale Shared-Memory Multiprocessors: Issues and Comparisons. *ACM Computing Surveys*, 25(3):303–338, September 1993.
- [20] Jeffrey B. Rothman and Alan Jay Smith. Analysis of Shared Memory Misses and Reference Patterns. Technical Report UCB/CSD-99-1064, Computer Science Division, University of California, Berkeley, September 1999.
- [21] Jeffrey B. Rothman and Alan Jay Smith. Minerva: An Adaptive Subblock Coherence Protocol for Improved SMP Performance. Technical Report UCB/CSD-99-1087, Computer Science Division, University of California, Berkeley, December 1999.
- [22] Jeffrey B. Rothman and Alan Jay Smith. Multiprocessor Memory Reference Generation Using Cerberus. In *Seventh International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS '99)*, pages 278–287, College Park, MD, October 24–28 1999.
- [23] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory. Technical report, Stanford University, June 1992. Report No. CSL-TR-92-526.
- [24] Per Stenström. A Survey of Cache Coherence Schemes for Multiprocessors. *IEEE Computer*, 23(6):12–25, June 1990.
- [25] Josep Torrellas, Monica S. Lam, and John L. Hennessy. False Sharing and Spatial Locality in Multiprocessor Caches. *IEEE Transactions on Computers*, 43(6):651–663, June 1994.
- [26] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proc. 22nd Annual International Symposium on Computer Architecture*, pages 24–36, Santa Margherita Ligure, Italy, June 22–24 1995.

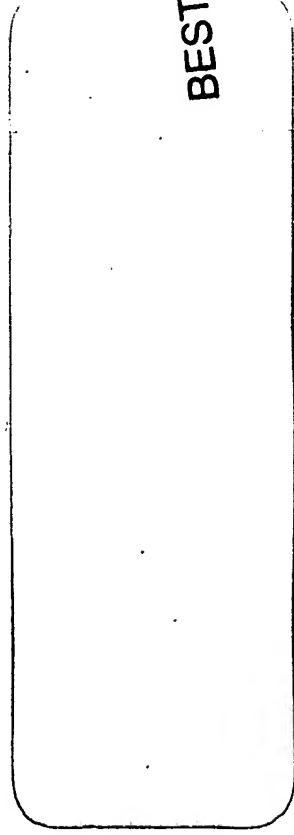


02 1A 0004204479 FEB 28 2006  
\$0.45  
MAILED FROM ZIP CODE 22314



AN EQUAL OPPORTUNITY EMPLOYER

RECEIVED  
MAR 22 2006  
USPTO MAIL CENTER



BEST AVAILABLE COPY

P.O. BOX 1450  
ALEXANDRIA, VA 22313-1450  
IF UNDELIVERABLE RETURN IN TEN DAYS  
OFFICIAL BUSINESS

**This Page is Inserted by IFW Indexing and Scanning  
Operations and is not part of the Official Record**

**BEST AVAILABLE IMAGES**

Defective images within this document are accurate representations of the original documents submitted by the applicant.

Defects in the images include but are not limited to the items checked:

- ☒ **BLACK BORDERS**
- ☐ **IMAGE CUT OFF AT TOP, BOTTOM OR SIDES**
- ☐ **FADED TEXT OR DRAWING**
- ☐ **BLURRED OR ILLEGIBLE TEXT OR DRAWING**
- ☐ **SKEWED/SLANTED IMAGES**
- ☐ **COLOR OR BLACK AND WHITE PHOTOGRAPHS**
- ☐ **GRAY SCALE DOCUMENTS**
- ☒ **LINES OR MARKS ON ORIGINAL DOCUMENT**
- ☐ **REFERENCE(S) OR EXHIBIT(S) SUBMITTED ARE POOR QUALITY**
- ☐ **OTHER:** \_\_\_\_\_

**IMAGES ARE BEST AVAILABLE COPY.**

**As rescanning these documents will not correct the image problems checked, please do not report these problems to the IFW Image Problem Mailbox.**